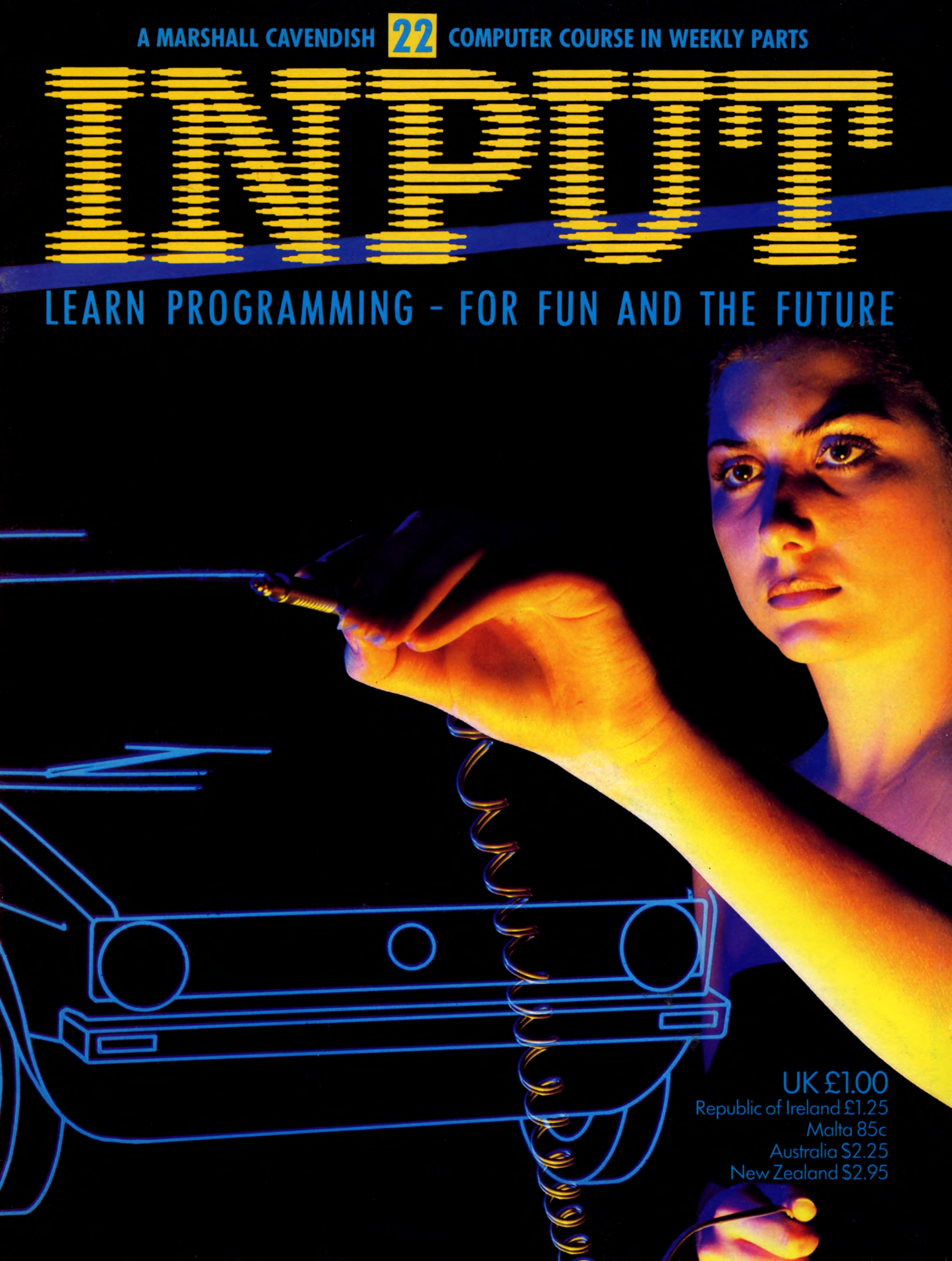


A MARSHALL CAVENDISH 22 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE
UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95



Vol. 2 **No 22**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Dave King. Pages 669, 670, Steve Bielschowsky/Berry Fallon Design. Page 672, Peter Reilly. Page 675, Sue Hillwood-Harris. The Sound of Music © Chappell Music Ltd. Pages 676, 678, 680, Gerry Banks. Page 683, Malcolm Livingstone. Pages 684, 686, 688, Kevin O'Keefe. Page 690, Dave King/Studio 10. Page 692, Dave King. Pages 694, 696, Jim Mawtus. Pages 697, 698, 699, Digital Arts.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN
Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015
New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington
Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).
UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services, Newtown Road, Hove BN3 7DN
Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

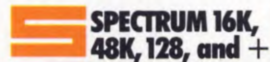
HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

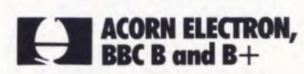
INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:




COMMODORE 64 and 128




DRAGON 32 and 64
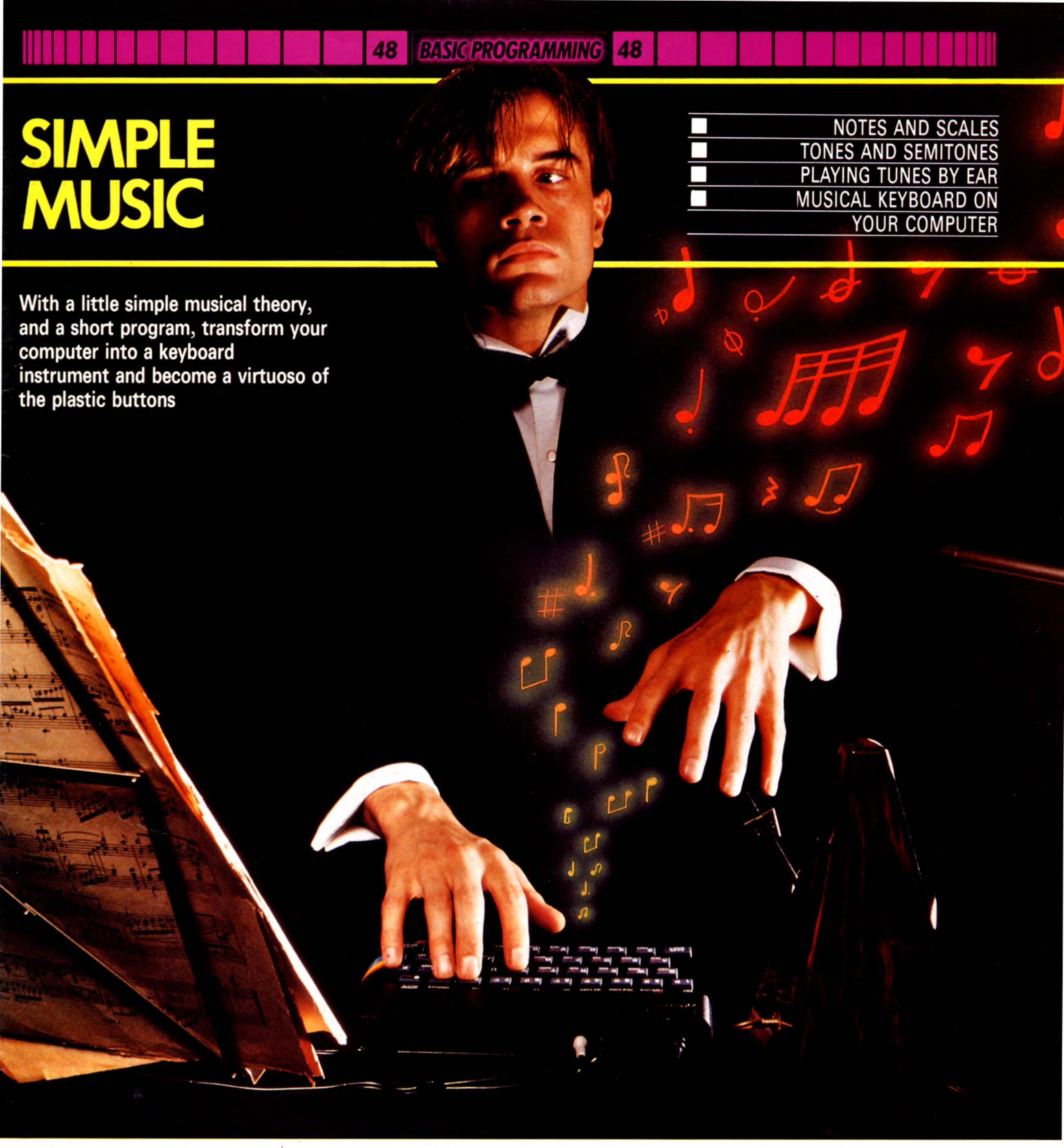

ZX81

VIC 20
TANDY TRS80 COLOUR COMPUTER

SIMPLE MUSIC

- NOTES AND SCALES
- TONES AND SEMITONES
- PLAYING TUNES BY EAR
- MUSICAL KEYBOARD ON YOUR COMPUTER

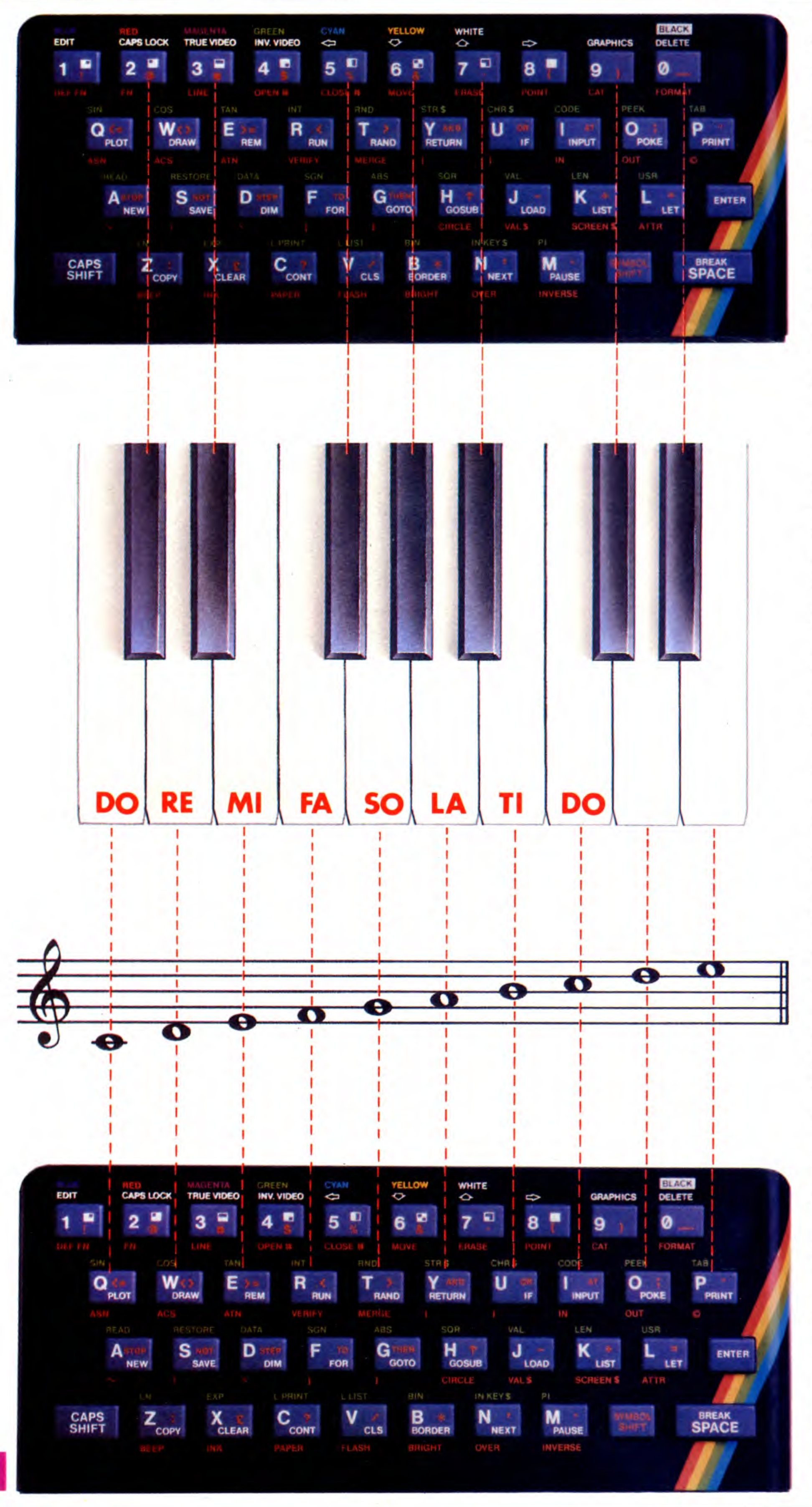
With a little simple musical theory, and a short program, transform your computer into a keyboard instrument and become a virtuoso of the plastic buttons



Each of the computers covered here, except the ZX81, has the facility to generate sound of some sort. The article on pages 230 to 235 explains how your computer's sound commands work. You can set the pitch, duration, and sometimes other aspects too, of the note that your computer plays.

The computers vary in several important respects. Most importantly, on the Spectrum, Dragon and Tandy, you only have the facility to play one note at a time—whereas the BBC and Commodores can play a chord of two or more notes simultaneously. Also on the Acorns and Commodores, you are able to change the whole quality of the sound. In some respects, the difference between these and the other computers is a bit like the difference between a mouth organ and a simple synthesizer.

But none of these differences matters at the moment. You can play simple tunes on any of the computers, and this article explains how to get started. It begins with a little very simple musical theory, so you will understand



the background, and then goes on to give a program which converts part of your computer's keyboard into a simple musical instrument. In a later article, you will see how to extend this to exploit more of your computer's special abilities.

But first, a couple of simple, fundamental definitions. 'Pitch' is the property of 'highness' or 'lowness' that sounds possess, so a musical note that sounds higher than a second note is said to have a higher pitch than it, and a note lower than another note has a lower pitch than it. And 'musical interval' is the 'musical distance' between two notes. The two notes at the beginning of 'Oh, I do like to be beside the seaside' have a small musical interval between them, as you shall see, while those at the beginning of 'Here comes the bride' have quite a large one: sing the first few notes of each to yourself, and notice the difference. (Start on the same note each time).

A SIMPLE SCALE

A scale is a series of notes, each higher than the last. While technically the notes are named after letters from A to G, it is often helpful to use the easily remembered labels **do**, **re**, **me**, **fa**, **so**, **la**, and **ti**. The scale starts at **do**, and goes up to the next **do**, going through 8 notes.

The **do-re-me** scale is known as the 'major scale', to put is slightly more technically. The major scale is defined by the particular set of musical relationships between its notes, rather than by its pitch; it's a kind of abstract arrangement of musical intervals which can be moved bodily to any pitch but will still retain its 'major' quality.

WHAT ABOUT THE BLACK KEYS

The white keys on a keyboard instrument are named A, B, C, D, E, F and G, then A, B, C and so on again, in a repeating cycle. The major scale is produced by playing the 8 white keys from a 'C' up to the next 'C', and this is, reasonably enough, called the scale of 'C major': C is **do**, D is **re** and so on. As you saw above, a major scale can start on any pitch, but C major starts on the note C, which has a particular pitch: choosing to start the scale on C 'anchors' the abstract major scale to a particular pitch.

What, then, are the black keys for? Well, the major scale doesn't include all possible notes between its bottom and top notes. Notes, not belonging to the scale, exist between **do** and **re**, and between certain other pairs of its notes: the black keys provide these notes. The 'black' note between C and D (**do** and **re** in C major) is called C sharp or D flat, ('sharp' simply means 'raised' and 'flat' means

'lowered'). It depends upon whether you base your definition on the note above, or below, the black note. The note between D and E is D sharp or E flat, and so on. Notice that there isn't a black key between E and F (that is, between **mi** and **fa** in C major), or between B and C (**ti** and **do** in C major) so no note exists between them.

The interval between any key and its immediate neighbour (whether a black or a white key) is called a semitone, so C to C sharp, C sharp to D, and E to F are all semitones. An interval containing 2 semitones, eg C to D, are known as a 'tone' or 'whole tone'. The arrangement of tones and semitones that defines a major scale is as follows: there are

2 semitones = 1 tone between **do** and **re**, (C and D in C major)
2 semitones = 1 tone between **re** and **mi**, (D and E in C major)
1 semitone between **mi** and **fa**, (E and F in C major)
2 semitones = 1 tone between **fa** and **so**, (F and G in C major)
2 semitones = 1 tone between **so** and **la**, (G and A in C major)
2 semitones = 1 tone between **la** and **ti**, (A and B in C major)
1 semitone between **ti** and **do**, (B and C in C major)

DIFFERENT MAJOR SCALES

So altogether there are 12 semitones from the bottom **do** to the top **do**, distributed among the 8 notes of the major scale as you can see above.

Suppose you want to construct a major scale on the note G, say, so that **do** will be **G**, **re** will be A and so on. If you just play the 8 white keys from a G up to the next G, the arrangement of intervals isn't right: the number of steps between **la** and **ti**, and between **ti** and **do**, aren't correct. In fact, you need to take F sharp rather than F as the **ti** in the new scale, then the arrangement of steps will be correct. You might like to check this on the diagram of the keyboard.

In a similar way, the major scale with F as a starting note has the notes F, G, A, B flat, C, D, E and F—B has to be flattened or the note **fa**, and the intervals **mi-fa** and **fa-so** will be wrong. In fact, you can construct a major scale on any note, if you sharpen or flatten the right notes, but any major scale built on a starting note other than C will necessarily use one or more black keys: only the major scale starting on C uses the white keys alone. This makes it the easiest scale to play in, and explains why it's so popular. You can refer to the notes either as C, D, E . . . C, or **do, re, mi** . . . **do**; this article uses the latter sequence, since it's more meaningful than the former one to most people.

Many, many tunes can be played using the major scale, in particular, folk songs, nursery tunes and some hymns. So, for example, 'Three Blind Mice' starts with the notes:

mi, re, do,
mi, re, do,
so, fa, fa, mi,
so, fa, fa, mi . . .

And the hymn 'Jerusalem' starts with

do, mi, so,
la, do,
la, so, fa, so

This just gives the relative pitches of the notes, of course, and not the rhythm; standard musical notation, which is not covered in this article, is a sophisticated system containing pitch, rhythm and also loudness and softness information.

More complicated pieces of music won't contain just the notes of the major scale and no others. Typically, they will start in one scale and, as the piece unfolds, will move temporarily into other scales which use some notes not part of the first scale; so for example, the hymn 'Oh God Our Help in Ages Past' starts:

so, mi, la, so, do, do, ti, do
so, do, so, la, fa sharp, so

because the second phrase makes a very brief visit to the major scale on a different note, before returning to the original scale. Notes foreign to the scale may also be used to make the tune more colourful. The first three notes of 'Oh, I do like to be beside the seaside' illustrate this: they are '**so**', '**so** sharp', and '**la**', and the song continues with **so, mi, re, do**.

WHY BOTH SHARPS AND FLATS?

Why do musicians use a sharp (F sharp) when creating G major, and a flat (B flat) with F major? F sharp and G flat are the same actual black key, after all. So why couldn't G flat be used in place of F sharp in G major, or A sharp in place of B flat in F major, thus getting rid of sharps entirely, or of flats entirely? You need both sharps and flats because it helps if each different note in the scale is identified by its own, different, letter name: to replace B flat by A sharp would turn the F major scale into F, G, A, A sharp, C, D, E and F, giving two As but no Bs, which could be confusing. And if you replaced the F sharp in G major by a G flat, you'd get two Gs (G flat and G) but no Fs. Using sharps where there should be flats and vice versa is a musical howler, like a spelling mistake in English, and music packages that let you use only sharps or only flats, arguing that C sharp is exactly equivalent to D flat, are inelegant.

FREQUENCY AND INTERVALS

Musical sounds are produced by repetitive vibrations in the air. The higher the rate of repetition, or 'frequency', of the vibration, the higher the pitch of the resulting note. The unit for measuring frequency is cycles-per-second (cps), or, more commonly, Hz (standing for hertz, and meaning the same thing).

If you take a note with a frequency of say 256 Hz (the frequency of the note C), and double its frequency to give 512 Hz, the second note is a C exactly an octave above the first. (An octave is the musical interval between the low **do** and the high **do** in the simple scale from **do** to the **do** above). A further frequency doubling, to give 1024 Hz, produces the next C, whose pitch is exactly an octave above the previous one, and so on. So *multiplying* the frequency of a note by some value *adds* a given musical interval to that note.

Now you've seen that there are 12 equal semitones in the octave (there are 13 keys, 8 white and 5 black, with 12 semitones *between* them). If doubling the frequency gives rise to the addition of a musical octave, and there are 12 semitones in an octave, then what multiplication of frequency will add the musical interval of a semitone? Well, with the octave we have

frequency of a note × 2 =
frequency of the note an octave above.

Let's call 'X' the multiplication required for a semitone: then we have

frequency
*X*X*X*X*X*X*X*X*X*X*X*X =
frequency of octave above.

So the number which divides the ratio 2:1 into 12 equal divisions in the correct way is the twelfth root of 2 (evaluated by the BASIC expression 2↑(1/12)). Multiplying 256 Hz by it adds a semitone, another multiplication adds a second semitone, and so on, until after 12 cumulative multiplications the frequency of the upper octave is reached. This is why the twelfth root of two is a fundamental constant in music, and crops up frequently in articles on music.

TUNES IN THE MAJOR SCALE

Suppose you want to play a tune on the simple keyboard programs later in the article. How

do you know which note is **do**, which is **re**, and so on, if you're not given this information? In other words, how can you play tunes 'by ear'? If you just start on a note at random, and try and fit the rest of the notes around it, the chances are you'll get it wrong.

Is there a method you can use to deduce how a tune fits into the major scale? It helps if you can find which note in the tune is **do**, so you can use this to deduce the position of the rest of the notes. Many tunes start on **do** (the hymn Jerusalem, for example), or finish on it (Three Blind Mice), or both start and finish on it (Pop Goes the Weasel). And **do** is generally the note in the tune which acts as a kind of musical focal point or its centre of gravity. Once you've found which note in the tune is **do** you can, with a bit of luck, find the rest of the notes. Try to play the tune from the beginning, note by note; you'll have to listen very carefully to what you're playing, deciding whether the tune goes up or down at each point, and whether it goes to a neighbouring note in the scale (which it often will), or skips a note, or 2 notes, or whatever.

If you make a mistake about which note of the tune is **do**, then some of the notes you play just won't sound right: they'll sound sharp or flat, and you'll have to re-assess which note is **do**. But if you persevere, this will become easier, you'll develop an intuition about scales, and eventually you'll be able to pick out any tune on a musical keyboard. But to start you off, this article gives the notes for several tunes.

KEYBOARD PROGRAMS

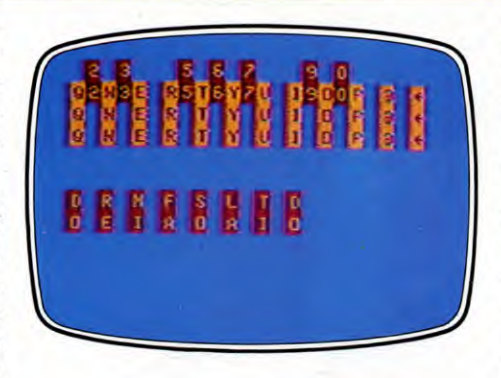
The programs all use the top two rows of the computer keyboard as a simple musical keyboard; 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I' will play **do**, **re**, **mi** and so on to **do**, and the keys to the right of these on the same row play in the next octave. The keys '2', '3', '5', '6', '7' act as the black notes, so the computer keys correspond to the layout of the musical keyboard in this way:

```
  2  3     5  6  7        etc
Q  W  E  R  T  Y  U  I    etc

d  r  m  f  s  l  t  d
o  e  i  a  o  a  i  o
```

The programs display this information constantly as a reminder while the program runs. In addition, when notes in the first octave of the major scale are being played, their names are displayed as the program runs.

Type in the program and RUN it. Following the listings there are some guidelines on how to become a keyboard virtuoso.



The musical keyboard as it appears on the Dragon and Tandy

The Spectrum program sets up the top two rows of the keyboard as a sort of musical instrument, so that when you press particular keys, an appropriate note is played.

The program begins by setting up variables for the duration of each note, the Line number of the main loop (this is stored as a variable loop, rather than an actual number 1000 to save memory), and for the PRINT AT position of part of the screen display. It then continues by setting up the screen display, and then starts the sound-generating part.

Loop is where the program returns after playing a note, and where the program waits for a key to be pressed. When a key is pressed, the CODE number of the character just pressed is put into the variable note.

The computer then goes to the Line number which is held in note. The line numbers correspond to the CODE numbers of the keys that the program uses (from 48 to 121). And the BEEP command in each line is set to play the appropriate note (for example, the Q key plays **do**). You will remember from the article on page 230 that the first number after BEEP gives the note's duration (here a variable, d), the second gives its pitch.

Some of the keys also print on the screen the name of the note they are playing—this happens with all the notes in the first octave of the major scale, and is done in the same line as that which plays the note.

When the note has been played, the computer returns to the main loop, at Line 1000, to wait for another keypress.

```
1 GOTO 900
47 GOTO loop
48 BEEP d,15: GOTO loop
50 BEEP d,1: GOTO loop
51 BEEP d,3: GOTO loop
53 BEEP d,6: GOTO loop
54 BEEP d,8: GOTO loop
55 BEEP d,10: GOTO loop
57 BEEP d,13: GOTO loop
101 PRINT AT y,x;"MI": BEEP d,4: GOTO loop
105 PRINT AT y,x;"DO": BEEP d,12: GOTO
   loop
111 BEEP d,14: GOTO loop
112 BEEP d,16: GOTO loop
113 PRINT AT y,x;"DO": BEEP d,0: GOTO
   loop
114 PRINT AT y,x;"FA": BEEP d,5: GOTO loop
116 PRINT AT y,x;"SO": BEEP d,7: GOTO loop
117 PRINT AT y,x;"TI": BEEP d,11: GOTO loop
119 PRINT AT y,x;"RE": BEEP d,2: GOTO loop
121 PRINT AT y,x;"LA": BEEP d,9: GOTO loop
800 GOTO loop
900 LET d=.03: LET x=5: LET
   loop=1000
901 BORDER 4: PAPER 4: CLS
902 PRINT AT 8,6;"d□r□m□
   f□s□l□t□d"
903 PRINT AT 9,6;"o□e□i□
   a□o□a□i□o"
910 LET a$="□2□3□□□
   5□6□7□□□9□0"
920 FOR y=3 TO 4: GOSUB 990
930 NEXT y
940 LET a$="Q□W□E□R□T
   □Y□U□I□O□P"
941 PAPER 7: INK 0
950 FOR y=4 TO 6: GOSUB 990
960 NEXT y
980 LET x=15: LET y=15: GOTO loop
990 FOR i=1 TO LEN a$
991 IF a$(i)< >CHR$ 32 THEN PRINT AT
   y,x+i;a$(i);
992 NEXT i:RETURN
1000 PRINT AT y,x;CHR$ 32;CHR$ 32
1005 LET a$=INKEY$: IF a$="" THEN
   GOTO 1000
1100 LET note=CODE a$: GOTO note
```

When you RUN this program and hold down a key for a long time you will hear a pulsating sound. This is because the Spectrum needs to BEEP for a fixed length of time. So what the program actually does is to play the same note over and over again, for a very short time.

The Commodore computers have quite complicated sound facilities, which need several POKEs to use. How you can do this, is explained on pages 232 and 233. Some of the POKEs need only be done once, to 'initialize' the sound chip—and these are done at the start of the program, in the subroutine starting at Line 3000. This sets, for example, the wavelength of each sound. (Since the Vic's sound works in a different way, this routine is not in the Vic program.)

The subroutine which starts at Line 4000

READs the data for the notes that are to be used into an array, so that the music-generating commands can access them easily. The subroutine from Line 6000 onwards sets up the screen display.

When the computer has GOne to all these SUB routines, it waits for you to press a key. When you do, the key's ASCII code is put into the variable X to be used to set the pitch of the note that is played in Lines 120 and 130. These two lines POKE the high byte and the low byte needed for each note, into memory. The Vic needs only one POKE to set the pitch, and this is done in Line 104.

The IF X$ = . . . lines check to see whether the key you have pressed has been assigned a name within the scale—and if it has, the computer prints it.

If you have pressed a key which does not play a note, all that happens is that the computer goes through the same routine, to play a note, but plays a nonexistent note—so no sound is heard.

Line 150 PEEKs the keyboard to see if the key is still being pressed—and if it is, the 'gate is held open' so that the sound carries on. This is how the continuous sound effect is achieved.

As soon as you are no longer pressing the key, the sound is stopped, the cursor sent back to the top left of the keyboard, and the computer returns to Line 100 to GET another keypress.

The values held in the array TA% for each note for the Vic do not correspond exactly with those in the Vic manual. The reason for this is that the Vic's sound chip has quite poor 'frequency resolution', which means that it cannot be exactly in tune. So the notes in this program are in tune **with each other**, but not with, say, a piano that is in 'concert pitch'.

```
40 GOSUB 3000
50 GOSUB 4000
70 GOSUB 6000
100 GET X$: PP=PEEK(197): IF X$=
   "" THEN 100
110 X=ASC(X$):POKE 198,0
111 IF X$="Q" OR X$="I" THEN
   PRINT "▤DO"
112 IF X$="W" THEN PRINT "▤RE"
113 IF X$="E" THEN PRINT "▤MI"
114 IF X$="R" THEN PRINT "▤FA"
115 IF X$="T" THEN PRINT "▤SO"
116 IF X$="Y" THEN PRINT "▤LA"
117 IF X$="U" THEN PRINT "▤TI"
120 POKE SI+4,33
130 POKE SI ,LQ%(X)
140 POKE SI+1,HQ%(X)
150 IF PP=PEEK(197) AND PP<>64
   THEN 150
170 POKE SI+4,32
180 PRINT "▤□□"
190 GOTO 100
3000 SI=54272
3010 FOR I=SI TO SI+28: POKE
   I,0:NEXTI
3020 POKE SI+5,16+11
3030 POKE SI+6,16*15+12
3040 POKE SI+24,4
3050 RETURN
4000 DIM HQ%(255), LQ%(255)
4010 TMP=4455
4020 FOR I=1 TO 22
4030 READ V$: V=ASC(V$)
4040 LQ%(V)=TMP-256*INT(TMP/
   256): HQ%(V)=TMP/256
4050 TMP=TMP*(2↑(1/12))
4060 NEXT: RETURN
4070 DATA Q,2,W,3,E,R,5,T,6,Y,7,U,I
4080 DATA 9,O,0,P,@,-,*,£,↑
6000 PRINT "◘": PRINT
6010 POKE 53280,8: POKE 53281,8
6020 BL$= "□□□□▦■2▮3
   ▮▮▮5▮6▮7▮▮▮9▮0
   ▮▮▮-▮£"
6030 PRINT BL$: PRINT BL$
6040 PRINT "◘";
6050 WH$= "□□□▦▤Q▮W
   ▮E▮R▮T▮Y▮U▮I▮O▮
   P▮@▮*▮↑"
6060 PRINT WH$: PRINT WH$:PRINT WH$
6065 PRINT
6070 PRINT"□□□□D□R□M□
   F□S□L□T□D"
6080 PRINT"□□□□O□E□I□
   A□O□A□I□O"
6090 RETURN
```

```
40 GOSUB 4000
50 GOSUB 6000
90 POKE 36878,0
100 GET X$: PP=PEEK(197):IFX$=""
   THEN 100
102 X=ASC(X$):POKE198,0
104 POKE 36876,TA%(X)
106 FOR V=1 TO 5: POKE 36878,V: NEXT
111 IF X$="Q" OR X$="I" THEN
   PRINT"▤DO"
112 IF X$="W" THEN PRINT"▤RE"
113 IF X$="E" THEN PRINT"▤MI"
114 IF X$="R" THEN PRINT"▤FA"
115 IF X$="T" THEN PRINT"▤SO"
116 IF X$="Y" THEN PRINT"▤LA"
117 IF X$="U" THEN PRINT"▤TI"
150 IF PP=PEEK(197) AND PP<>64
   THEN 150
180 PRINT"▤□□"
185 FOR V=4 TO 0 STEP-1: POKE
   36878,V: NEXT
190 GOTO 100
4000 DIM TA%(255)
4010 FOR I=1 TO 22: READ V$,V:
   TA%(ASC(V$))=255-V: NEXT
4020 RETURN
4030 DATA Q,90,2,85,W,80,3,76,E,72,
   R,67,5,64,T,60,6,57,Y,54,7,51,U,48,I,45
4040 DATA 9,42,O,40,0,38,P,36,@,34,
   -,32,*,30,£,28,↑,27
6000 PRINT "◘": PRINT
6010 POKE 36879,127
6020 BL$="□▦■2▮3▮▮▮5▮
   6▮7▮▮▮9▮0"
6030 PRINT BL$; PRINT BL$
6040 PRINT "◘";
6050 WH$="▦▤Q▮W▮E▮R▮
   T▮Y▮U▮I▮O▮P"
6060 PRINT WH$: PRINTWH$: PRINTWH$
6070 PRINT"■"
6080 PRINT"D□R□M□F□
   S□L□T□D"
6090 PRINT"O□E□I□A□
   O□A□I□O"
6100 RETURN
```

Microtip

You can make the Spectrum's quiet BEEP sound louder in a number of ways. Several companies sell add-ons to channel the sound through an external amplifier and loudspeaker. But you do not need to spend any money at all.

If your tape recorder has a monitoring facility, you can use its speaker. First, connect the MIC leads between the Spectrum and the tape recorder, and then press RECORD and PLAY on your recorder. After this, any sound will come out of the loudspeaker on your tape recorder. An advantage of this, is that you can now control the tone and volume of the sound.

This does, though, have a low background noise, from the recorder's motor. If you find this too loud, an add-on is the only answer.



Lines 5 and 6 set the autorepeat to work faster than usual to make the program work better. The values of the frequency for each note are stored in the array TA by Lines 10 to 20. After this, Lines 30 to 51 print up the keyboard in red background and black ink.

The main routine of the program starts at Line 100, with the computer waiting for you



Why doesn't the Dragon and Tandy keyboard program have auto-repeating keys?
The Dragon and Tandy program uses INKEY$ to check the keyboard. It would be possible to use PEEK, which is the usual way to provide an auto-repeat, but the computer would need to PEEK about 20 numbers, one for each key. And because each number has no logical numeric connection with the next (since the keys are QWE . . . and not ABC . . .) these would need to be in a DATA statement which takes a long time to check. It's quicker to press each key twice.

to press a key. The next Line, 11Ø, actually plays the note, using the SOUND command (see page 233 to see how this works and what each number following it means).

The third number in this SOUND command sets the pitch. The function ASC returns the ASCII code of a character—here the key that you have just pressed. The number that this function gives is then used to tell the computer which number in the array to look at—and so which note to play (TA holds the pitch for each note that the program uses).

If you have pressed a key which is not used by the program, this is detected by Line 1Ø5. The value of TA for every key that is not used is Ø so all this line does is to check whether this value is Ø or not. When the pitch is Ø, the computer goes back to wait for another keypress. The reason for this is so that the computer does not try to play a note with a pitch of Ø—try it, and you'll see why.

Once the computer has played a valid note, it carries on and prints up the name of the note it has just played in the top left-hand corner of the keyboard. It first prints up two spaces, and then prints up the **so**, or whatever name applies to the note; if the note has not been assigned a name (only the notes in the main octave actually have names in this program) nothing is printed.

The computer then returns to the start of the main loop to wait for another keypress. Electron changes are at the end.

```
5 *FX11,12
6 *FX12,15
1Ø DIM TA(255)
15 FOR I=1 TO 22
2Ø READ V$,V: TA(ASC(V$))=V:NEXT
3Ø MODE 1: COLOUR 129: COLOUR Ø: VDU
   12
35 BL$="□□□□□□2□3
   □□□5□6□7□□□9□
   Ø□□□∧□\"
36 PRINT: PRINT: PRINT BL$: PRINT BL$
4Ø COLOUR 3
45 WH$="□□□□□Q□W□E
   □R□T□Y□U□I□O□P□@
   □[□—"
46 PRINT WH$: PRINT WH$: PRINT WH$
48 COLOUR 2:PRINT,
5Ø PRINT "□□□□□D□R□M
   □F□S□L□T□D"
51 PRINT "□□□□□O□E□I
   □A□O□A□I□O"
1ØØ A$=GET$: IF A$="" THEN 1ØØ
1Ø5 IF TA(ASC(A$))=Ø THEN 1ØØ
11Ø SOUND 1,-15,TA(ASC(A$)),3
115 VDU 31,15,13:PRINT"□□": VDU 31,
   15,13
12Ø IF A$="Q" OR A$="I" THEN PRINT
   "DO"
13Ø IF A$="W" THEN PRINT "RE"
14Ø IF A$="E" THEN PRINT "MI"
15Ø IF A$="R" THEN PRINT "FA"
16Ø IF A$="T" THEN PRINT "SO"
17Ø IF A$="Y" THEN PRINT "LA"
18Ø IF A$="U" THEN PRINT "TI"
2ØØ GOTO 1ØØ
1ØØØ DATA Q,1Ø1,2,1Ø5,W,1Ø9,3,113,E,117
1Ø1Ø DATA R,121,5,125,T,129,6,133,Y,137
1Ø2Ø DATA 7,141,U,145,I,149,9,153,O,157
1Ø3Ø DATA Ø,161,P,165,@,169,∧,173,[,177
1Ø4Ø DATA \,181,_,185
```

For the Electron delete Line 1Ø4Ø and alter:

```
15 FOR I=1 TO 17
35 BL$="□□□□□□2□3□□□
   5□6□7□□□9□Ø"
45 WH$="□□□□□Q□W□E□R
   □T□Y□U□I□O□P"
1Ø3Ø DATA Ø,161,P,165
```

The Dragon and Tandy program starts by clearing the screen in blue, and setting up the volume, note length, and tempo for the PLAY commands which the program uses. Page 235 explains how the string in Line 1Ø sets these parameters.

Lines 2Ø and 3Ø set up the notes which the program uses into the array N$. Line 4Ø sets up a string to hold the various possible keys for each note, and one to assign a name (**do**, **re**, **me**, and so on) to some of the notes.

The lines 5Ø to 11Ø set up the screen display, and sets the screen colours to give orange and black.

Then the computer waits for a key to be pressed; if the key that is pressed is not set to play a note, the computer returns to Line 12Ø to wait for another keypress.

Line 14Ø actually plays the note, using the PLAY command. The pitch is set by making the variable NT dependent upon which key has been pressed, and using the character NT spaces into the array N$ as the pitch.

Then, using an ON . . . GOTO . . . statement, the computer prints the name of the note it has just played (if a name has been assigned to that note) before returning to wait for your next keypress.

```
1Ø CLS3:PLAY"V31L4T8":B$=CHR$(175)
2Ø DIM N$(21):FORK=ØTO19:READ
   N$(K):NEXT
3Ø DATA C,C#,D,D#,E,F,F#,G,G#,A,
   A#,B,O4C,O4C#,O4D,O4D#,O4E,
   O4F,O4F#,O4G
4Ø M$="Q2W3ER5T6Y7UI9OØP@—"+
   CHR$(8):D$="QWERTYUIO"
5Ø FORK=ØTO6:READA:N=2*A-2Ø*
   INT(A/1Ø):POKE1Ø93+N,A:POKE1125
   +N,A:NEXT
6Ø DATA 5Ø,51,53,54,55,57,48
65 POKE1113,45:POKE1145,45
7Ø FORK=ØTO11:READA:POKE1124+
   K*2,A:POKE1156+K*2,A:POKE
   1188+K*2,A:NEXT
8Ø DATA 81,87,69,82,84,89,85,73,
   79,8Ø,64,95
9Ø PRINT@26Ø,"d"B$"r"B$"m"B$
   "f"B$"s"B$"l"B$"t"B$"d";
1ØØ PRINT@292,"o"B$"e"B$"i"B$
   "a"B$"o"B$"a"B$"i"B$"o";
11Ø SCREENØ,1
12Ø A$=INKEY$:IF A$="" THEN12Ø
13Ø NT=INSTR(M$,A$):IF NT=Ø THEN12Ø
14Ø PLAY"O3"+N$(NT-1)
15Ø PRINT@Ø,;:D=INSTR(D$,A$):
   ON D GOTO 17Ø,18Ø,19Ø,2ØØ,
   21Ø,22Ø,23Ø,17Ø
16Ø PRINTB$;B$;:GOTO11Ø
17Ø PRINT"DO";:GOTO11Ø
18Ø PRINT"RE";:GOTO11Ø
19Ø PRINT"MI";:GOTO11Ø
2ØØ PRINT"FA";:GOTO11Ø
21Ø PRINT"SO";:GOTO11Ø
22Ø PRINT"LA";:GOTO11Ø
23Ø PRINT"TI";:GOTO11Ø
```

TICKLING THE IVORIES

So what can you play with this musical keyboard? Try the nursery rhyme 'Pop Goes the Weasel'. This starts on the note **do**, so the first key will be 'Q'; you'll need to experiment with the rhythm until it sounds right:

Q,E,W,R,E,T,E,Q
Q,E,W,R,E,Q
Q,E,W,R,E,T,E,Q
Y,W,R,E,Q

And now the famous bit from Beethoven's Ninth Symphony (try it and you'll recognize it, even if you don.t know the name):

E,E,R,T,T,R,E,W
Q,Q,W,E,E,W,W
E,E,R,T,T,R,E,W
Q,Q,W,E,W,Q,Q
W,E,Q,W,E,R,E,Q
W,E,R,E,W,Q,W,T
E,E,R,T,T,R,E,W
Q,Q,W,E,W,Q,Q

Now the song from 'The Sound of Music':

Q,W,E,Q,E,Q,E
W,E,R,R,E,W,R
E,R,T,E,T,E,T
R,T,Y,Y,T,R,Y
T,Q,W,E,R,T,Y
Y,W,E,5,T,Y,U
U,E,5,6,Y,U,I
I,U,Y,R,U,T,I

There are just three notes foreign to the scale towards the end.

Finally, here are the keys you should press to play the beginning of the Bach piece 'Jesu, Joy of Man's Desiring':

Q,W,E,T,R,R,Y,T,T,I,U,I,T,E
Q,W,E,R,T,Y,T,R,E,W,E,Q

Unfortunately, at this point the tune goes off the end of the keyboard. But you can play it all in the major scale starting on **fa**:

R,T,Y,I,7,7,O,I,I,@,P,@,I,Y
R,T,Y,7,I,O,I,7,Y,T,Y,R,E,R,T
Q,E,T,7,Y,T,Y
R,T,Y,I,7,7,O,I,I,@,P,@,I,Y
R,T,Y,W,I,7,Y,T,R,Q,R,E,R

(The '@' indicates the key next to 'P' on the Commodore, BBC, Dragon and Tandy keyboards. Unfortunately as there is no equivalent key next to the P on the Spectrum and Electron, this note is not available and these computers cannot play this tune.)

Try your hand at 'Three Blind Mice'. To get you going; here are the first few lines:

E,W,Q,E,W,Q
T,R,R,E,T,R,R,E
T,I,I,U,Y,U,I,T,T
T,I,I,U,Y,U,I,T,T



COMMODORE DISK DRIVE CONVERTER

Commodore programs that SAVE to and LOAD from tape can be made to work with a disk drive after using this simple machine code converter routine

So far most programs given in *INPUT* have assumed that you are SAVEing to and LOADing from tape. But more and more people are buying disk units which are a faster and more efficient way of SAVEing your programs.

Of course, if you do have a disk unit you can modify the programs given in *INPUT* and elsewhere to work with it, by hand, yourself. But why not let your computer do it for you? The following assembly language program will modify tape-dependent programs for use with disk drives on the Commodore 64 and Vic 20.

The BBC Micro does not need a conversion program as it will default to disk drive if one is present. The Electron and ZX81 don't have disk drive units. It is not possible to give one for the Dragon as there are three different disk systems available. And a Microdrive conversion program for the Spectrum was given last time.

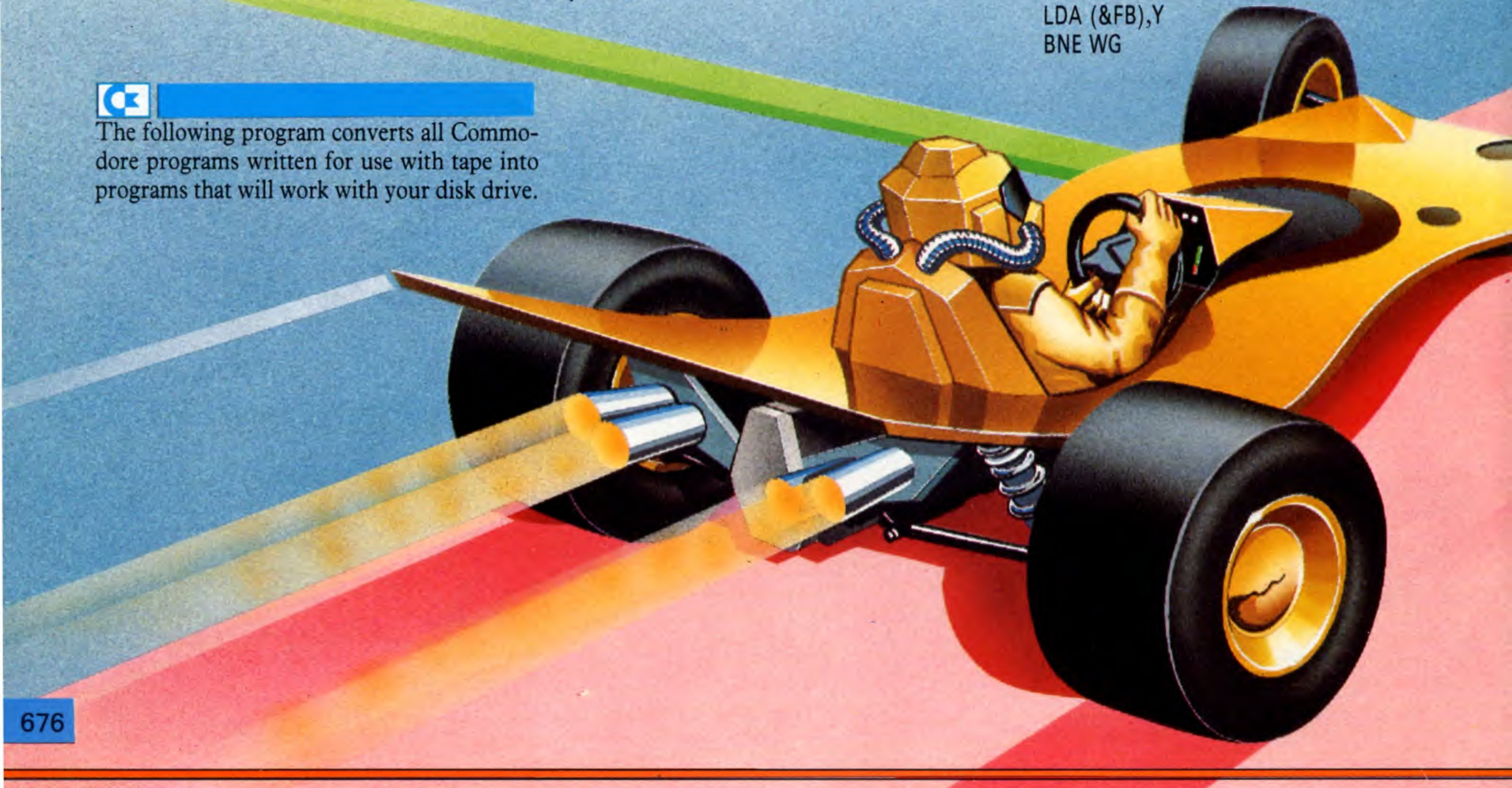
Essentially all it has to do is add ,8 after any tape command. 8 is the device number for the first disk drive attached. But this program goes a little further than that. There is a difference SAVEing on disk rather than tape. If you SAVE on tape you can overwrite one program with another, or save a program or file as many times as you like under the same name. But you cannot SAVE a program or file on disk twice with the same name. If you try, you'll get a disk error message.

Programs with names prefixed by @:□ will overwrite any previous version on the disk. So this utility also adds @:□ before the name of any program.

However, program names are sometimes stored as strings. To have this disk utility go back over the program, find the appropriate string, and add @:□ to it, would double the length of the routine. So this eventuality has not been covered. If one of your programs does store file names this way you have no alternative but to add @:□ within the quotes at the beginning of the program name yourself.

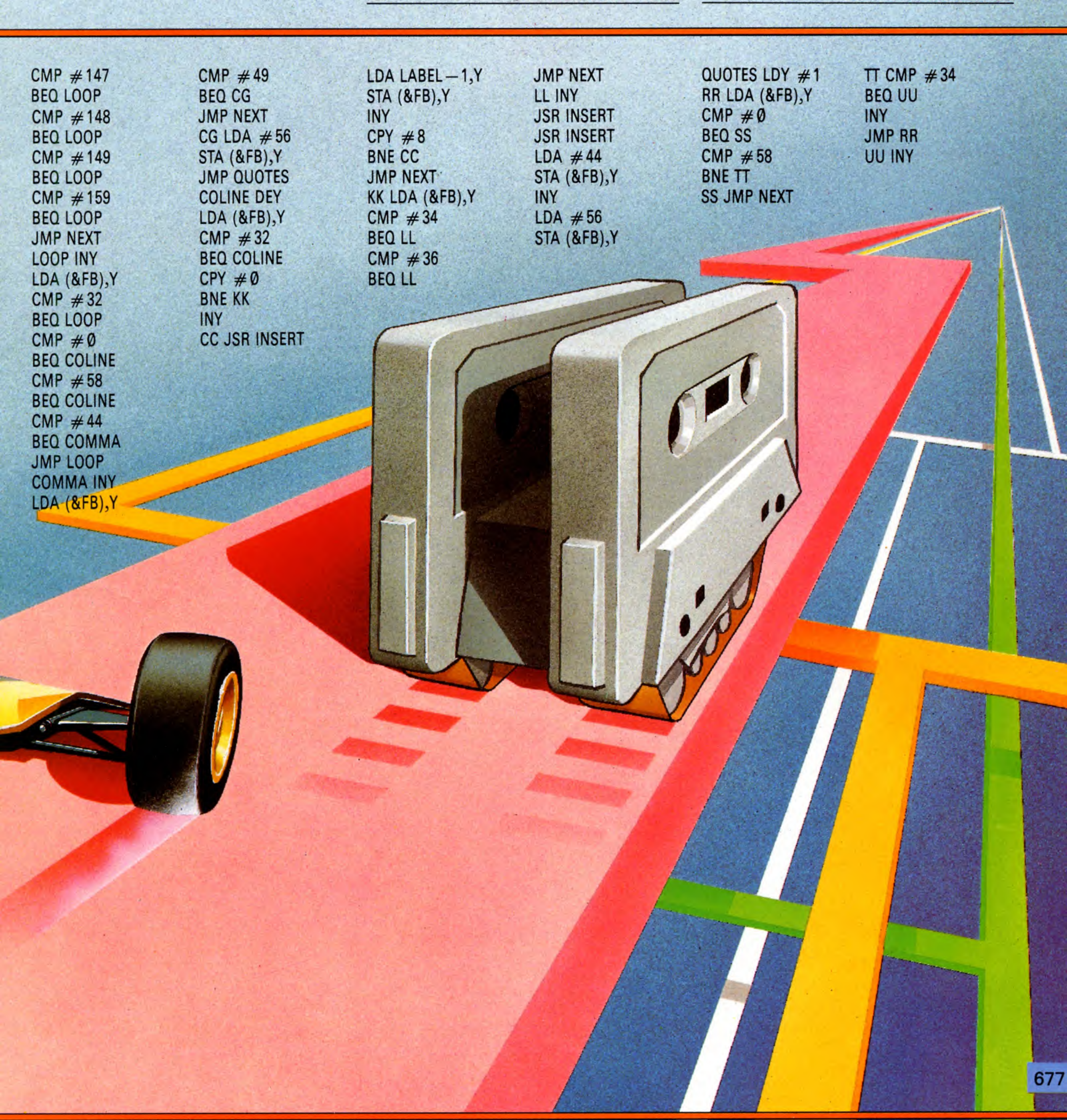
```
ORG□49152
LDA #Ø
STA &Ø33E
CLC
LDA &2B
ADC #3
STA &FB
LDA &2C
ADC #Ø
STA &FC
NEXT LDY #Ø
INC &FB
BNE AA
INC &FC
AA LDA &FB
CMP &2D
BNE BB
LDA &FC
CMP &2E
BNE BB
JMP STOP
BB LDA (&FB),Y
CMP #Ø
BNE A
INY
LDA (&FB),Y
BNE WG
INY
LDA (&FB),Y
BNE WG
JMP STOP
WG LDA &FB
ADC #3
STA &FB
LDA &FC
ADC #Ø
STA &FC
AD LDX #Ø
STX &Ø33E
JMP NEXT
AB CMP #34
BNE AC
LDX &Ø33E
CPX #Ø
BNE AD
LDX #1
STX 3Ø33E
JMP NEXT
AB LDX &Ø33E
CPX #1
BEQ NEXT
```

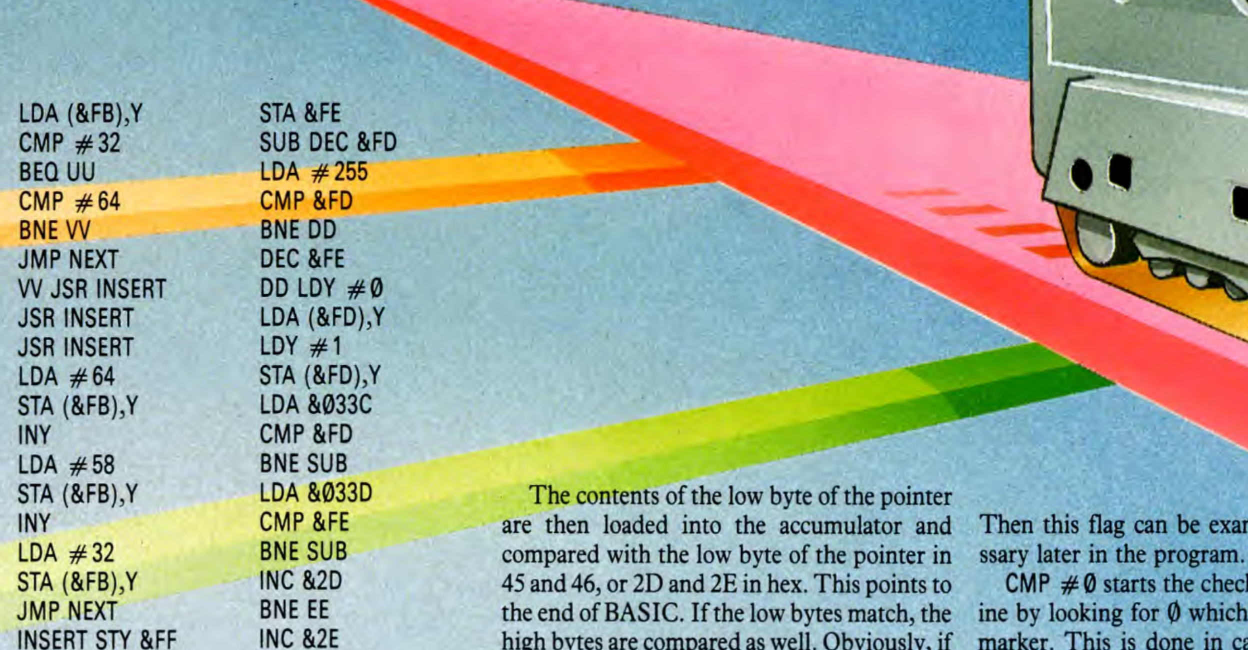
The following program converts all Commodore programs written for use with tape into programs that will work with your disk drive.



- LOCATING **SAVES**, **LOADS**, **OPENS** AND **VERIFYS**
- AVOIDING KEYBOARD GRAPHIC COMMANDS
- OVERWRITING FILES
- USING A DATA TABLE
- MOVING THE PROGRAM UP
- ADDING DATA
- UPDATING THE SYSTEM VARIABLES

```
CMP #147
BEQ LOOP
CMP #148
BEQ LOOP
CMP #149
BEQ LOOP
CMP #159
BEQ LOOP
JMP NEXT
LOOP INY
LDA (&FB),Y
CMP #32
BEQ LOOP
CMP #Ø
BEQ COLINE
CMP #58
BEQ COLINE
CMP #44
BEQ COMMA
JMP LOOP
COMMA INY
LDA (&FB),Y
CMP #49
BEQ CG
JMP NEXT
CG LDA #56
STA (&FB),Y
JMP QUOTES
COLINE DEY
LDA (&FB),Y
CMP #32
BEQ COLINE
CPY #Ø
BNE KK
INY
CC JSR INSERT
LDA LABEL—1,Y
STA (&FB),Y
INY
CPY #8
BNE CC
JMP NEXT
KK LDA (&FB),Y
CMP #34
BEQ LL
CMP #36
BEQ LL
JMP NEXT
LL INY
JSR INSERT
JSR INSERT
LDA #44
STA (&FB),Y
INY
LDA #56
STA (&FB),Y
QUOTES LDY #1
RR LDA (&FB),Y
CMP #Ø
BEQ SS
CMP #58
BNE TT
SS JMP NEXT
TT CMP #34
BEQ UU
INY
JMP RR
UU INY
```





```
LDA (&FB),Y          STA &FE
CMP #32              SUB DEC &FD
BEQ UU               LDA #255
CMP #64              CMP &FD
BNE VV               BNE DD
JMP NEXT             DEC &FE
VV JSR INSERT        DD LDY #Ø
JSR INSERT           LDA (&FD),Y
JSR INSERT           LDY #1
LDA #64              STA (&FD),Y
STA (&FB),Y          LDA &Ø33C
INY                  CMP &FD
LDA #58              BNE SUB
STA (&FB),Y          LDA &Ø33D
INY                  CMP &FE
LDA #32              BNE SUB
STA (&FB),Y          INC &2D
JMP NEXT             BNE EE
INSERT STY &FF       INC &2E
CLC                  EE LDY &FF
LDA &FC              RTS
STA &Ø33D            STOP JSR &A533
PP TYA               RTS
ADC &FB              LABEL BYT &22
BCC QQ               BYT &4Ø
INC &Ø33D            BYT &3A
QQ STA &Ø33C         BYT &2Ø
LDA &2D              BYT &22
STA &FD              BYT &2C
LDA &2E              BYT &38
```

HOW IT WORKS

The first thing the program does is load Ø into the accumulator and store it in memory location Ø33E. This clears a byte that is going to be used as a flag later in the routine.

The pointer to the start of BASIC is kept in memory locations 43 and 44, or 2B and 2C in hex. The first four instructions of this routine store this pointer in the zero page memory locations FB and FC which are in the user area of the zero page. There you can manipulate them.

Then the Y register is loaded with the number Ø. INC &FB INCrements the contents of FB, in other words moving the pointer onto the next byte of the BASIC program. If the result is zero when this is incremented, the following Branch if result Not Equal command, BNE AA, does not operate—and the high byte of the pointer is incremented. This way, the transition across the edge of a page is automatically accounted for.

The contents of the low byte of the pointer are then loaded into the accumulator and compared with the low byte of the pointer in 45 and 46, or 2D and 2E in hex. This points to the end of BASIC. If the low bytes match, the high bytes are compared as well. Obviously, if both bytes match, this routine has reached the end of the BASIC program—so it jumps to STOP at the end of the routine.

Otherwise, the byte pointed to by FB and FC offset by the value carried in the Y register is loaded into the accumulator by LDA (&FB),Y. You will note that the value carried in the Y register is always Ø at this point, so there is no offset. But indirect addressing is needed at this point and all the indirect addressed instructions are indexed.

CHECKING FOR QUOTES

Two of the Commodore keyboard graphics symbols, clear screen and insert, can be confused with the tape commands, LOAD and SAVE. The ASCII codes for the two graphs symbols are the same as the tokens for the commands. But there is one way to tell the difference between them. The graphics symbols always appear in quotes, while SAVE and LOAD never do. Even if the words SAVE and LOAD appear in quotes, they are not commands and are not tokenized. They are stored as ASCII characters so that they can be PRINTed on the screen.

So the routine has to check whether the byte it is dealing with is in quotes or not. How you do this is set a flag—in this case the memory location Ø33E that was cleared at the beginning of the program. The flag should be set when the first quote in a line is encountered. And when the second quote in a line is encountered you have to reset it again. Then this flag can be examined when necessary later in the program.

CMP #Ø starts the check-for-quotes routine by looking for Ø which is the end of line marker. This is done in case there has only been one quote mark in a line. If Ø is found, BNE AB does not make the branch, Ø is loaded into the X register and stored in Ø33E, clearing the flag. Then JMP NEXT takes the processor back to start checking the next byte of the BASIC program.

If Ø is not found, the processor branches to the label AB and compares the byte with 34, the ASCII value of quote marks. Again if one is found the BNE does not make the branch. The contents of memory location Ø33E is loaded into the X register and compared with Ø. If it is zero, the BNE AD does not make the branch, 1 is loaded into the X register and stored back in Ø33E. If it's not zero—that is, if it has previously been set to 1—the BNE AD takes the processor back to the routine which stores Ø in Ø33E to clear it. Either way, it meets a JMP NEXT, which starts the routine off on the next byte of BASIC again.

However, if no quote is found, the BNE AC takes the processor on to LDX &Ø33E which loads the contents of Ø33E into the X register. This is compared to 1 to see whether it is set. If it is, BEQ NEXT takes the processor back to start on the next byte again. If not, it goes on to check for the tape commands.

FINDING TAPE COMMANDS

The next byte of the BASIC program is still in the accumulator, and CMP compares it with the number 147, the token for LOAD. If it is 147, the following Branch if EQual, BEQ LOOP, sends the processor to the LOOP label.



If the byte is not the token for LOAD, it is compared with 148, the token for SAVE, 149, the token for VERIFY, and 159, the token for OPEN. If none of these tape commands is found, JMP NEXT sends the processor back again to the beginning of the routine to deal with the next byte. But if any of these tokens are found, the processor is sent off the LOOP routine.

THE LOOP ROUTINE

INY increments the Y register and LDA (&FB),Y loads the next byte after the tape command into the accumulator. The routine then uses CMP #32 to check if it is a space. 32 is the ASCII for a space.

If the next character is a space, BEQ LOOP sends the processor back to the beginning of the LOOP routine again to see whether the following character is a space as well.

Otherwise, CMP #Ø checks for a Ø end of line marker. If it is found, BEQ COLINE takes the processor off to the COLINE routine.

If it's not a space or an end of line marker, the byte is compared with 58, the ASCII code for a colon. If it is a colon, BEQ COLINE branches to the COLINE routine.

The next thing that is checked for is a comma. CMP #44 compares the byte with the ASCII for a comma (44). If the next byte is a comma, BEQ COMMA sends the processor off to execute the COMMA routine. If not, JMP LOOP jumps back to the beginning of the LOOP routine, so that the next byte can be checked for a space, end of line, colon or comma.

THE COMMA ROUTINE

If the loop routine found a comma, it sends the processor off to the COMMA routine. This immediately increments the contents of the Y register and LDA (&FB),Y loads the accumulator with the contents of the byte following the comma.

This is compared with 49, the ASCII for 1. The number 1 after an output command indicates device number 1—a tape recorder—is to be used. The number 2 would indicate the screen, 4 a printer, 8 a disk drive. So if what follows is not a 1, the routine branches right back to the beginning of the routine again.

If the next byte is the ASCII for 1, 56—the ASCII for 8—is loaded into the accumulator by LDA #56. This is then stored in the appropriate location by STA (&FB),Y. The processor jumps to the QUOTES routine.

THE COLINE ROUTINE

The first thing the COLINE routine does is decrement the Y register. Then it loads the accumulator with the contents of the byte immediately before the colon or end of line marker. If this is a space, CMP #32 and BEQ COLINE sends it back to do it again. In other words, the routine moves back from the colon or end of line marker until it bumps into a character that is not a space. What it is looking for is a program name in quotes.

When it has found something, a character that is not a space, CPY #Ø checks to see whether the Y register contains Ø, which

would mean that it has gone all the way back to the tape command. If it has, BEQ KK takes the processor to the KK routine.

JSR INSERT Jumps to the INSERT SubRoutine which moves the rest of the BASIC program up one byte in memory, so that something can be inserted. Then the accumulator is loaded with data from the LABEL table at the end of the program.



Remember, this part of the program is only executed if there is nothing following the tape command, except spaces and a colon or end of line marker. So Y starts off as Ø. It is then incremented to point to the first byte past the command, and LDA LABEL−1,Y loads the accumulator with the first byte in the LABEL table. This byte is then stored in the BASIC program. INY increments Y. CPY #8 compares the contents of the Y register with 8. If Y hasn't clocked up as far as 8, BNE CC sends the processor back to move the BASIC program up again and store another byte from the table.

The seven bytes of data in the LABEL table are 22, 4Ø, 3A, 2Ø, 22, 2C and 38 which are the hex for the ASCII codes of the characters making up "@:□",8. This turns the simple tape command LOAD, for example, into LOAD "@:□",8.

When these seven bytes have been inserted the Y register is incremented to 8 and the BNE CC instruction does not make the branch and JMP NEXT sends the processor back to the beginning of the program to start searching for the next tape command.

THE KK ROUTINE

If, when the routine was checking back from the colon or end of line marker, it found a character other than the token for the tape save command, the KK routine is called which checks for quotes or dollar signs.

So the first thing it does is load up the byte in question. CMP #34, BEQ LL and CMP #36, BEQ LL checks for quote marks or a dollar sign and takes the processor to LL if either occur. If the byte contains a character that is neither of these, it is an illegal character under these circumstances. So the routine ignores it and jumps back to NEXT to check the next character.

THE LL ROUTINE

If the KK routine has picked up a quote mark or a dollar sign it branches to the LL routine. This immediately increments the Y register to look at the next byte, then moves the rest of the BASIC program up two bytes to leave a two-byte space, by calling the INSERT subroutine twice.

What needs to be filled into the two spaces left empty here is ,8. This transforms the tape commands SAVE "progname" or SAVE P$, or whatever you've called your program, into the disk commands SAVE "progname",8 or SAVE P$,8.

So the accumulator is loaded with 44, the ASCII for a comma, which is then stored in the appropriate byte by the STA (&FB),Y. Then the Y register is incremented and the accumulator loaded with 56, the ASCII for 8, so that the same STA (&FB),Y instruction loads the 8 into the next byte.

THE QUOTES ROUTINE

The QUOTES routine starts by setting Y back to 1, with LDY #1, then it loads up the first byte after the tape-instruction token with LDA (&FB),Y. This is then compared to Ø and 58 to check when it has reached the end of the BASIC statement in question again.

A rather peculiar construction is used here with BEQ SS branching to an instruction with an unconditional jump, JMP NEXT. This is used because the branch instruction will not branch more than 128 bytes backwards, or 127 bytes forward. Branch instructions only carry a one-byte offset while a JMP has a two-byte offset.

The next thing that is tested for is a quote mark, and if CMP #34 finds one BEQ UU branches on to the UU routine. If a quote mark is not found, the contents of Y are incremented and JMP RR loops to check the next byte. So the processor goes round and round this loop looking at the BASIC program a byte at a time until it hits an end of line marker, such as a colon or a quote mark.

Although the Y register is incremented either way, the CMP #34 test turns out a single INY before the branch cannot be used instead of the two used after the branch here, thereby saving a byte. INY affects the zero flag, which the BEQ UU uses when deciding which way to branch.

The next byte is then compared to 32, the ASCII for a space. If it is a space, the routine loops and the Y register is incremented again until the check hits something other than a space.

The next thing that is checked for is a @. Obviously, if the program name already has a @:□ to allow it to be overwritten, you don't want to add another one. So if the next byte is 64—ASCII for @—the processor jumps back to NEXT. Otherwise, it moves on to VV and calls the INSERT routine three times, moving all of the rest of the BASIC up three bytes in memory.

Then the numbers 64, 58 and 32 are stored in the three byte space created. These three numbers are the ASCII codes for @: and a space respectively. Once that has been done, JMP NEXT takes the processor back to the beginning again to check the rest of the BASIC program.



THE INSERT ROUTINE

The insert routine has been called several times during the machine code program so far. What it does is shift all the BASIC program following the point reached up in memory one byte.

The first thing it does is to store the contents of the Y register in the zero-page location FF. The contents of the Y register must be the same after you come out of the INSERT routine as they were when you entered it, otherwise you will lose your place in the program. But during the INSERT routine you are going to need the Y register. So its contents are saved at the beginning and, you will notice, the contents of FF are loaded back into the Y register at the end.

The carry flag is then cleared by the CLC, ready for the additions coming up. The value of the high byte of the pointer stored in FC is then copied into memory location Ø33D by LDA &FC and STA &Ø33D.

The contents of the Y register—which counts how many bytes past a tape instruction the program has reached—is transferred into the accumulator so that it can be added to the low byte of the pointer in FB and FC, which carries the address of the tape-instruction token itself. So the result of the addition gives the address of the last byte of BASIC program that is not going to be shifted up memory.

The result is stored in Ø33C whether it overflows the eight-bit accumulator or not. But if it does, the carry flag is set, the BCC—Branch Carry Clear—does not make the branch, and INC &Ø33D increments the high byte of the pointer.

The system variable that points to the start of the variables area—that is one byte after the end of the BASIC program—is carried by 2D and 2E. This is copied into FD and FE so that its value can be used.

After the label SUB there is a little routine which decrements the pointer in FD and FE. The first time this is done it decrements the pointer so that it points to the last byte of the BASIC program. But this little routine is going to be used over and over again to move byte by byte down the BASIC program.

The Y register is loaded with Ø and the accumulator is loaded with the contents of the memory location pointed to by the contents of FD and FE, offset by Y. Y is then incremented by 1 and STA (&FD),Y stores the byte back again one location further up memory.

The pointer you've stored in Ø33C and Ø33D which tells you where you are in the BASIC program is compared with the value in FD and FE. If they are not the same and the routine has not worked its way all the way down to where the insert is to be made, the routine branches back to the label SUB where the pointers are decremented and the next byte is shifted.

When both bytes of the pointers match, all the BASIC program after the insert has been shifted. The only thing left to do in the routine is to update the end of BASIC/start of variables area system variable. This is done by INC &2D, BNE EE and INC &2E.

The LDY &FF restores the value of the Y register and RTS returns to the instruction after the routine was called.

TIDYING UP

By moving the BASIC program about in memory you have messed up all the pointers which point to the beginning of each line of BASIC. These are stored at the beginning of the preceding line of BASIC.

Luckily there is a ROM routine which corrects them all. So all you have to do is call the subroutine at A533 with JSR &A533. RTS then returns the computer to BASIC.

What follows is not strictly assembly language. It is data. The BYT is an assembler directive which tells it to set aside one byte of memory for the data which follows.

To call the program use:

```
SYS 49152
```

Don't forget to enter a BASIC program that needs converting first though, otherwise calling the machine code routine will be a waste of time. And if you do forget, the routine will crash.

To SAVE this program you have to use your machine code monitor (see page 280). But if you want to use the monitor to SAVE to disk instead of tape and try to convert it with this program, you are going to run into some difficulty. The machine code monitor is an unusual program because the SAVE option is accessed from a print statement. The SAVE is not actually done by the program itself. You do it when you move the cursor to SAVE on the screen and press RETURN. So the monitor is one of the programs you will have to amend by hand.

This is easy enough to do. You simply have to add ,8 to the end of Line 33Ø. So Line 33Ø should read:

```
330 PRINT"■■P┌46,"BB":P┌45,"
   B2:PRINT"■■SAVE"CHR$(34)
   N$CHR$(34)",8"
```

And when you LOAD the machine code program back in off tape you must suffix the LOAD instruction with a ,8,1 as in:

```
LOAD "PROGNAME",8,1
```

The ,8 tells your computer that a disk drive is being used and the ,1 tells the machine to put the code back in the same place in memory it was taken from. Then type NEW to reset the BASIC pointers.

The assembly language program given above will work on Vic 20 as well. But you will have to relocate it as what follows 49152 is not a protected area. Try using 7168 as your origin but don't forget to alter the pointers to shift RAMTOP first to protect it. To do that use the following POKEs:

```
POKE 51,255
POKE 52,27
POKE 55,255
POKE 56,27
```

Then:

```
CLR
```

The ROM that puts the BASIC line pointers right after all the shifting has been down at the end of the routine is different too. It starts at C533 so that instruction should read JSR &C533.

If you don't have a commercial assembler for your Vic 20 you can enter the program via your machine code monitor (see page 280). If you don't fancy hand assembling it here is the hex machine code which you can enter directly:

```
A9 ØØ 8D 3E Ø3 18 A5 2B 69 Ø3 85 FB A5 2C
63 ØØ 85 FC AØ ØØ E6 FB DØ Ø2 E6 FC A5
FB C5 2D DØ Ø9 A5 FC C5 2E DØ Ø3 4C 6Ø
1D B1 FB C9 ØØ DØ 21 C8 B1 FB DØ Ø8 C8
B1 FB DØ Ø3 4C 6Ø 1D A5 FB 69 Ø3 85 FB
A5 FC 69 ØØ 85 FC A2 ØØ 8E 3E Ø3 4C 12 1C
C9 22 DØ ØF AE 3E Ø3 EØ ØØ DØ ED A2 Ø1
8E 3E Ø3 4C 12 1C AE 3E Ø3 EØ Ø1 FØ A8 C9
93 FØ ØF C9 94 FØ ØB C9 95 FØ Ø7 C9 9F FØ
Ø3 4C 12 1C C8 B1 FB C9 2Ø FØ F9 C9 ØØ FØ
1C C9 3A FØ 18 C9 2C FØ Ø3 4C 7D 1C C8
B1 FB C9 31 FØ Ø3 4C 12 1C A9 38 91 FB 4C
DD 1C 88 B1 FB C9 2Ø FØ F9 CØ ØØ DØ 11
C8 2Ø 1C 1D B9 63 1D 91 FB C8 CØ Ø8 DØ
F3 4C 12 1C B1 FB C9 22 FØ Ø7 C9 24 FØ Ø3
4C 12 1C C8 2Ø 1C 1D 2Ø 1C 1D A9 2C 91 FB
C8 A9 38 91 FB AØ Ø1 B1 FB C9 ØØ FØ Ø4 C9
3A DØ Ø3 4C 12 1C C9 22 FØ Ø4 C8 4C DF
1C C8 B1 FB C9 2Ø FØ F9 C9 4Ø DØ Ø3 4C 12
1C 2Ø 1C 1D 2Ø 1C 1D 2Ø 1C 1D A9 4Ø 91
FB C8 A9 3A 91 FB C8 A9 2Ø 91 FB 4C 12 1C
84 FF 18 A5 FC 8D 3D Ø3 98 65 FB 9Ø Ø3 EE
3D Ø3 8D 3C Ø3 A5 2D 85 FD A5 2E 85 FE C6
FD A9 FF C5 FD DØ Ø2 C6 FE AØ ØØ B1 FD
AØ Ø1 91 FD AD 3C Ø3 C5 FD DØ E7 AD 3D
Ø3 C5 FE DØ EØ E6 2D DØ Ø2 E6 2E A4 FF 6Ø
2Ø 33 C5 6Ø 22 4Ø 3A 2Ø 22 2C 38
```

To call the program use: SYS 7168

Don't forget to enter a BASIC program that needs converting first though, otherwise calling the machine code routine will be a waste of time. If you do forget, the routine will crash.

To SAVE this program you have to use your machine code monitor. But if you want to use the monitor to SAVE to disk you are going to run into some difficulty. The machine code monitor is an unusual program because the SAVE option is accessed from a print statement. The SAVE is not actually done by the program itself. You do it when you move the cursor to SAVE on the screen and press RETURN. So the monitor is one of the programs you will have to amend by hand.

This is easy enough to do. You simply have to add ,8 to the end of Line 33Ø. So Line 33Ø should read:

```
330 PRINT"■■P┌46,"BB":P┌45,"
   B2:PRINT"■■SAVE"CHR$(34)
   N$CHR$(34)",8"
```

And when you LOAD the machine code program back in off tape you must suffix the LOAD instruction with a ,8,1, as in:

```
LOAD "PROGNAME",8,1
```

The ,8 tells your computer that a disk drive is being used and the ,1 tells the machine to put the code back in the same place in memory it was taken from. Then type NEW to reset the BASIC pointers.



There is another circumstance under which the disk converter program won't work. It is when the program name has been stored as a string, as in:

```
SAVE P$
```

The program will add the ,8 after the dollar sign, so this instruction will work once. But as the actual program name is stored elsewhere in a line like:

```
P$="DATAFILE"
```

the routine can add the @:□ before the name, to allow it to be overwritten.

So when the program tries to overwrite a file, you will get an error message.

CARING FOR TAPES AND DISKS

- PROTECTING THE INFORMATION
- TAKING BACK-UPS
- INDEXING
- POSTING TAPES AND DISKS

Even if you're the sort of person who keeps important information on the backs of old envelopes, storing tapes and disks is one area where it doesn't pay to be disorganized



Storage systems based on magnetic tapes or disks make it possible for the home computer user to keep vast amounts of information, or thousands of programs, in a very compact form.

But the efficiency of magnetic storage systems is also a potential weakness. Because one small disk or tape can contain so much information, any damage may have disastrous results. And magnetic media are particularly vulnerable.

SECURITY

Obviously, the most important thing is to make sure that once you have put information on to tape or disk, it stays there—and that you can get it back when you want to. There are two sorts of damage that can affect magnetic media—physical and magnetic.

Physical damage can be anything from bending a disk or crushing a tape, to getting dirt on the surface. Prevent this by protecting them at all times when they are not actually in use. Always keep disks in their sleeves, and store them in a proper disk box. Keep tapes in their boxes—preferably in a rack (which also makes them easier to find). Store well away from heat, damp or dust.

Never touch the surface of a disk, or the exposed section of tapes. Rewinding tapes after use is good practice, since it means that only the less vulnerable leader is exposed, and also the tape is ready for use next time. Don't leave tapes in the recorder for a long time with the machine left on PLAY, as the head can kink the exposed section. Similarly, don't leave disks in the drive unit.

Magnetic fields strong enough to corrupt a stored file are put out by many pieces of household equipment. Loudspeakers contain strong magnets, and so do some electric motors even when they are not running—but there are less obvious sources. In general keep tapes—and especially disks—well away from any electrical equipment, including the television.

BELT AND BRACES

Accidents will happen, so it's worth keeping back-ups or taking extra precautions—especially for your most important files.

In general, it's a good idea to record two versions of everything, even on the same tape or disk. There's less likelihood of both being lost than just one being corrupted. But if the file is very important, make a copy on another tape or disk. Keep this one somewhere else—and don't use it. It can be a good idea to take tape back-ups of important disks—tape is less prone to corruption.

Protect your tapes from overwriting by removing the two knock-out sections from the back edge. Disks can be locked to prevent overwriting.

KEEPING TRACK

Tapes and disks build up over a long period, and it can be very hard to remember exactly what you put where. So the general rule is to label things as much as you can. Economy permitting, keep each file on a separate, short tape—or reserve disks for related files only.

Give each file a clear, unique name (within the confines of what your system permits). In particular, if you have several developments of one program, give each a new name (or number). Write the file names on the cassette label as well as the inlay card—tape can get separated from the case. Don't write on disk labels with a hard pen—this can score the disk. Keep a file index book to list all the names, where the file is, and any notes.

On the file it's worth including a REM statement with a date and description.

SENDING FILES

Tapes and disks are a very convenient way to send information or programs through the post. Tapes are reasonably strong and travel well—surprisingly, the case is more likely to suffer, so remove this and save weight, too. You can get special tape posting wallets, but for most purposes a padded bag is perfectly adequate.

The biggest risk with disks is bending, so pack them between two stiff cards.

Mark the outside of the package 'MAGNETIC MEDIA HANDLE WITH CARE'. For extra assurance, you can send it recorded delivery, or even registered mail.

USING YOUR TEXT COMPRESSOR

Sharpen up your pencil and get to work on your masterpiece. With a fully-functioning text compressor, you can write the epic adventure you've always dreamed about



By now you should have a machine code program stored on tape. The program will be exactly the same whether you have started from the assembly language listing, or the hexadecimal. But you will need to write some BASIC in order to use the machine code.

In this part of Games Programming you'll be shown how to use BASIC with the machine code text compressor, and how to harness the compressor to an adventure game.

DEVELOPING ADVENTURE GAMES

Earlier in *INPUT* you were shown how to write adventure games. With only a few alterations to the programs, you can use the text compressor to save a large amount of memory.

Initially, write the adventure according to the framework given earlier, but only include short messages, because they will be removed later. For example, you may want to say something along the lines of YOU HAVE ARRIVED IN A LEAFY GLADE. THERE IS A WOODEN LEG PARTIALLY HIDDEN BY A TABLECLOTH. Instead of entering this in full, use something like LEAFY in the PRINT statement. Do this with each of the PRINT statements so you can test that the adventure actually works, iron out bugs and make sure that there aren't any glaring logical mistakes.

Once you are satisfied with the game itself, you can move on to use the text compressor and rewrite the PRINT statement.

The first thing to do is to work out what your messages are actually going to say. If you have a printer, the task may be a little simpler as you can list the program, and work alongside the listing. Write out a list of messages, with the line numbers to which the messages correspond. Any PRINT statement with more than two or three words, is worth encoding. Very short messages will save little or no memory space by being encoded, so you may want to save the work of encoding them.

- SAVING MEMORY WITH THE TEXT COMPRESSOR
- ENCODING PRINT STATEMENTS
- HOW THE DECODING ROUTINE WORKS
- ADAPTING YOUR ADVENTURE PROGRAM
- MERGING THE ADVENTURE AND CODING PROGRAMS
- CHECKING THE MEMORY

WHAT'S HAPPENING

As you encode your text, the machine code program stores the binary in an array Z or Z%, depending on which machine the program has been written for. Another array, A or A%, contains pointers which enable the machine to keep track of where in Z or Z% each of the messages start.

When you want to decode—that is, to print the coded messages on screen—the pointers in A or A% are used to pick out the right section of binary for the decoding routine to work on. As the binary is decoded, the characters are fed into a string—Z$—which has been previously defined as being the length of the longest message. To make the message appear on the screen, all you need do is to tell the computer to PRINT Z$—and give it the appropriate number from A or A% that tells it which part of the message to feed into Z$.

Having tested the adventure program containing the short PRINT statements, you can start using the text compressor.

With the adventure game still in memory, LOAD or type in these program lines. See page 339 for how to merge programs.

```
9900 CLEAR 64580: LOAD""CODE
9905 LET mem=60000-(PEEK
   23653+256*PEEK 23654)
9910 PRINT "About□";mem;"□bytes free"
9915 DIM z (INT (mem/5))
9920 INPUT "How many messages",n
9925 DIM a(n)
9930 RANDOMIZE USR 64600
9935 LET z(1)=11
9940 FOR m=1 TO n
9945 CLS
9950 INPUT INVERSE 1;"Enter message
   number□";(m), LINE z$
9955 IF z$="" THEN LET m=n: GOTO 9975
9960 RANDOMIZE USR 64607
9965 LET a(m)=z(1)
9970 IF mem+z(1)<1100 AND
   mem+z(1)>900 THEN PRINT "About
   1000 bytes left"
9975 NEXT m
9980 PRINT FLASH 1; TAB 6;"Now save your
   program";TAB 6;"Do NOT RUN or
   CLEAR!";TAB 31;"□"
```

Type GOTO 9900 after positioning the tape containing the text compressor in your tape recorder. The machine code will be LOADed by Line 9900.

The program looks at how much memory remains, and prompts you for the number of messages you wish to enter. When you have entered this, the program will prompt you to type in each message in turn. You must be careful about entering messages which extend over more than one line. Do not use a carriage return for formatting this, because the computer will interpret it as an ended message. Instead, fill the lines up with spaces so that the messages format correctly—spaces don't use up too much memory space.

The program warns you when there is about 1000 bytes of memory remaining. If you receive this message when typing in, be careful that the program doesn't crash because parts of the programs overwrite each other. If you think you will run out of memory space, look at your list of messages for ways to shorten them, and start entering them again from the beginning.

When you've finished, there's a prompt to tell you to SAVE the program. Just type SAVE "name of program", and the adventure and the compressed text will be SAVEd together.

With the compressed text safely on tape, you can change the adventure program so that it can use the machine code text compressor and the stored data instead of the temporary PRINT statements. Delete all the lines from 9900 onwards, and type in this short subroutine instead:

```
9900 LET Z(1)=A(N):RANDOMIZE USR
   65067:PRINT Z$
9910 RETURN
```

Now go through the program, substituting a new instruction in place of each PRINT statement for which you have encoded a message. Suppose you wish to call the coded message for the second PRINT statement. Change

PRINT "message"

to

LET N=2:GOSUB 9900

This sets the variable N equal to the number of the message and then goes to the subroutine. This picks the Nth element of the array A, the pointers which select the right part of Z, loads the message into Z$ and prints it out.

Finally, you'll need a dummy Z$, for the machine code to use so that it can display the decoded text. As near to the start of the program as possible, define Z$ as anything you like, but it must be long enough to accept the longest message. Something along the lines of:

```
1 LET Z$="XXXXXXXXXXXXXetc."
```

With the new version of the program completed SAVE it again on tape. When you want to play the game, type

CLEAR 64580
LOAD""CODE (loading the text compressor)
LOAD"" (adventure plus compressed text)

Never press RUN to run the program with the compressed text, because you'll lose all of your messages. Always type GOTO 1 instead.

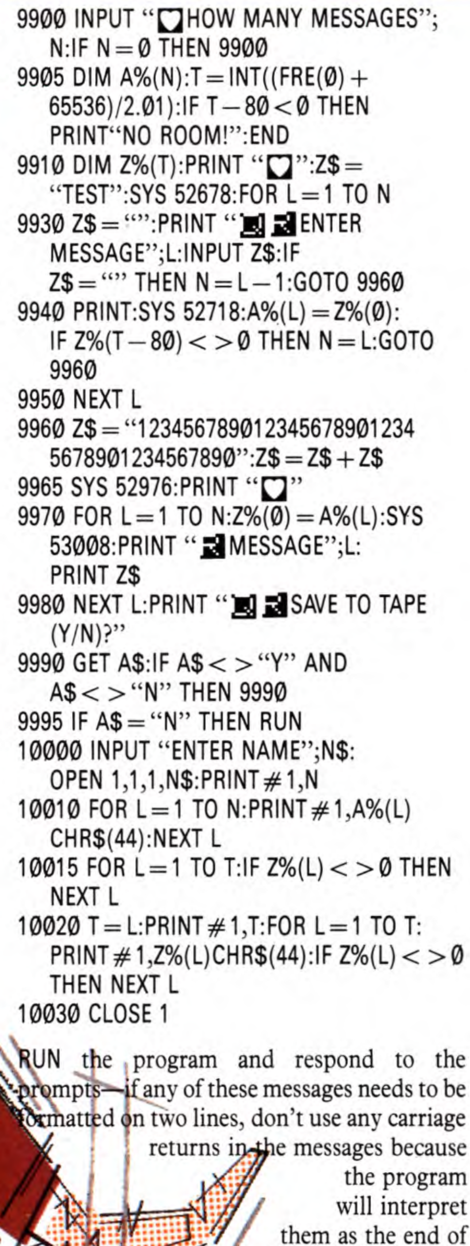



Once you have the adventure containing the shortened messages functioning to your satisfaction, you can enter these lines. The SYS calls are for the Commodore 64, but these are the numbers which you will need to substitute for the Vic 20 with various expansions:

Commodore 64	Vic 20 +8K	+16K	+24K	+32K
52718	15854	24046	32238	36334
52678	15814	24006	32198	36294
52976	16112	24304	32496	36592
53008	16144	24336	32528	36624

Also for the Vic, change Line 9905 to:

```
9905 DIM A%(N):T=INT
     (FRE(0)/2.01):
     IF T-80<0 THEN
     PRINT "NO
     ROOM!":
     END
```



```
9900 INPUT "[CLR]HOW MANY MESSAGES";
   N:IF N=0 THEN 9900
9905 DIM A%(N):T=INT((FRE(0)+
   65536)/2.01):IF T-80<0 THEN
   PRINT"NO ROOM!":END
9910 DIM Z%(T):PRINT "[CLR]":Z$=
   "TEST":SYS 52678:FOR L=1 TO N
9930 Z$="":PRINT "[HOME][DOWN]ENTER
   MESSAGE";L:INPUT Z$:IF
   Z$="" THEN N=L-1:GOTO 9960
9940 PRINT:SYS 52718:A%(L)=Z%(0):
   IF Z%(T-80)<>0 THEN N=L:GOTO
   9960
9950 NEXT L
9960 Z$="12345678901234567890l234
   567890l234567890":Z$=Z$+Z$
9965 SYS 52976:PRINT "[CLR]"
9970 FOR L=1 TO N:Z%(0)=A%(L):SYS
   53008:PRINT "[DOWN]MESSAGE";L:
   PRINT Z$
9980 NEXT L:PRINT "[HOME][DOWN]SAVE TO TAPE
   (Y/N)?"
9990 GET A$:IF A$<>"Y" AND
   A$<>"N" THEN 9990
9995 IF A$="N" THEN RUN
10000 INPUT "ENTER NAME";N$:
   OPEN 1,1,1,N$:PRINT#1,N
10010 FOR L=1 TO N:PRINT#1,A%(L)
   CHR$(44):NEXT L
10015 FOR L=1 TO T:IF Z%(L)<>0 THEN
   NEXT L
10020 T=L:PRINT#1,T:FOR L=1 TO T:
   PRINT#1,Z%(L)CHR$(44):IF Z%(L)<>0
   THEN NEXT L
10030 CLOSE 1
```

RUN the program and respond to the prompts—if any of these messages needs to be formatted on two lines, don't use any carriage returns in the messages because the program will interpret them as the end of the message. Instead fill up the lines with spaces—spaces do not waste too much memory.



Once you have completed entering the text, you'll be prompted to SAVE it to tape. Delete the lines from 99ØØ onwards, and add these lines to the start of the program:

```
1Ø CLR:INPUT "ENTER NAME";N$:
   OPEN 1,1,Ø,N$:INPUT#1,N:DIM A%(N)
2Ø FOR L=1 TO N:INPUT#1,A%(L):NEXT L
3Ø INPUT#1,T:DIM Z%(T):FOR L=1 TO
   T:INPUT#1,Z%(L):IF Z%(L)<>Ø THEN
   NEXT L
4Ø CLOSE 1
5Ø Z$="1234567890123456789Ø12345
   678901234567890":Z$=Z$+Z$
6Ø SYS 52976
```

Add this to the program, too:

```
9899 END
99ØØ Z%(Ø)=A%(L):SYS 53ØØ8:PRINT Z$:
   RETURN
```

Don't forget to change the SYS for the Vic.

Each of the PRINT statements for which you have compressed text must now be changed into a GOSUB 99ØØ. For example, if Line 5ØØ contains the fifth message, change:

```
5ØØ PRINT "SWAMP"
```

to

```
5ØØ L=5:GOSUB 99ØØ
```

L is a variable which is set equal to the number of the message. In the subroutine it picks the Lth element of the array A%, the pointers which select the right part of Z%. When you come to play the game, LOAD the text compressor itself into the memory first. Enter:

```
LOAD"name of text compressor",1,1
```

Then NEW the computer.

Next, LOAD the adventure and RUN it. It will take care of LOADing the compressed text, but make sure that you have the tape containing the compressed text ready.

As soon as you've debugged your adventure program containing the brief PRINT statements, and are satisfied with the game, you can think about using the text compressor.

The first step is to find out how much memory remains after the adventure program is in memory—type PRINT (?2+?3*256). You'll need this figure a little later, so write it down.

LOAD or type in the program below, and RUN it:

```
1Ø MODE6:HIMEM=&6ØØØ-&2ØØ:PRINT
   "LOOKING FOR ENCODE":*LOAD
   "ENCODE"
2Ø INPUT"WHAT WAS THE VALUE OF
   MEMORY LOCATIONS 2+256*3□",A$
3Ø INPUT"HOW MANY PIECES OF TEXT DO
   YOU WANT TO□□HAVE□",A: IF A<1
   THEN 3Ø
4Ø B=INT((&6ØØØ-&2ØØ-EVAL(A$)-
   A*4-&3ØØ)/4)+1
5Ø DIM a%(A+1),z%(B):z$=STRING$
   (255,"□")
6Ø ESTRING=HIMEM+174:CALL
   HIMEM+161
7Ø PRINT"ENTER YOUR MESSAGES NOW"
8Ø T=Ø:REPEAT
9Ø T=T+1
1ØØ INPUTLINE z$:IF z$="" THEN
   z$="□"
11Ø a%(T)=USR(ESTRING)AND&FFFF
12Ø PRINT"YOU HAVE□";B*4-a%(T);
   "□BYTES LEFT"
13Ø UNTIL (A=T OR a%(T)>B*4-4Ø OR
   z$="□")
14Ø IF z$="□" THEN 16Ø
15Ø z$="□":a%(T+1)=USR(ESTRING)
   AND&FFFF
16Ø INPUT"FILENAME PLEASE□",A$:
   C=LEN(A$):IF C<1 OR C>8 THEN
   PRINT"TOO LONG":GOTO 16Ø
17Ø H=OPENOUT(A$)
18Ø PRINT#H,A,B,T
19Ø FOR P=1 TO T:PRINT#H,a%(P):
   NEXT
2ØØ X=(a%(T+1) DIV 4)+1:PRINT#H,X
21Ø FOR P=1 TO X:PRINT#H,z%(P):NEXT
22Ø CLOSE#H
23Ø END
```

RUNning the program will cause the coder to be LOADed from tape. Make sure you have the tape positioned correctly and the PLAY button on the tape machine is depressed.

Enter the amount of memory remaining in response to the prompt; also the number of messages you have noted down. As you enter each message, there's an indicator showing how much memory remains. Be careful not to run out of memory, or the program will crash when you RUN it. If you do get close to running out of memory, the messages will have to be shortened, and entered from the start again.

When you've completed the encoding process, the program will prompt you for a filename for the encoded text, and SAVE it on tape or disk.

LOAD the adventure back into the machine, and add these lines to the start of the program. You may have to RENUMBER the adventure to accommodate the lines.

```
1Ø MODE6:HIMEM=&6ØØØ-&2ØØ:PRINT
   "LOOKING FOR DECODE":*LOAD
   "DECODE"
2Ø *OPT1,1
3Ø *OPT2,1
4Ø z$=STRING$(255,"□")
5Ø INPUT"WHAT IS THE NAME OF YOUR
   DATA PROGRAM",A$
6Ø IF LEN(A$)>8 THEN 5Ø
7Ø H=OPENIN(A$)
8Ø INPUT#H,A,B,T
9Ø DIM a%(A),z%(B)
1ØØ FOR P=1 TO T:INPUT#H,a%(P):NEXT
11Ø INPUT#H,X
12Ø FOR P=1 TO X:INPUT#H,z%(P):NEXT
13Ø CLOSE#H
```

```
140 CALL HIMEM+116
150 DSTRING=HIMEM+3Ø8
```

Add this PROCedure to the end of the program:

```
2ØØØØ DEF PROCWORD(N)
2ØØ1Ø z%(Ø)=a%(N):CALL DSTRING:
    PRINTz$:ENDPROC
```

Finally, the PRINT statements will have to be replaced by PROCedure calls. For example Line 5ØØ may contain the fifth message:

```
5ØØ PRINT "CASTLE"
```

You'd change it to:

```
5ØØ PROCWORD (5)
```

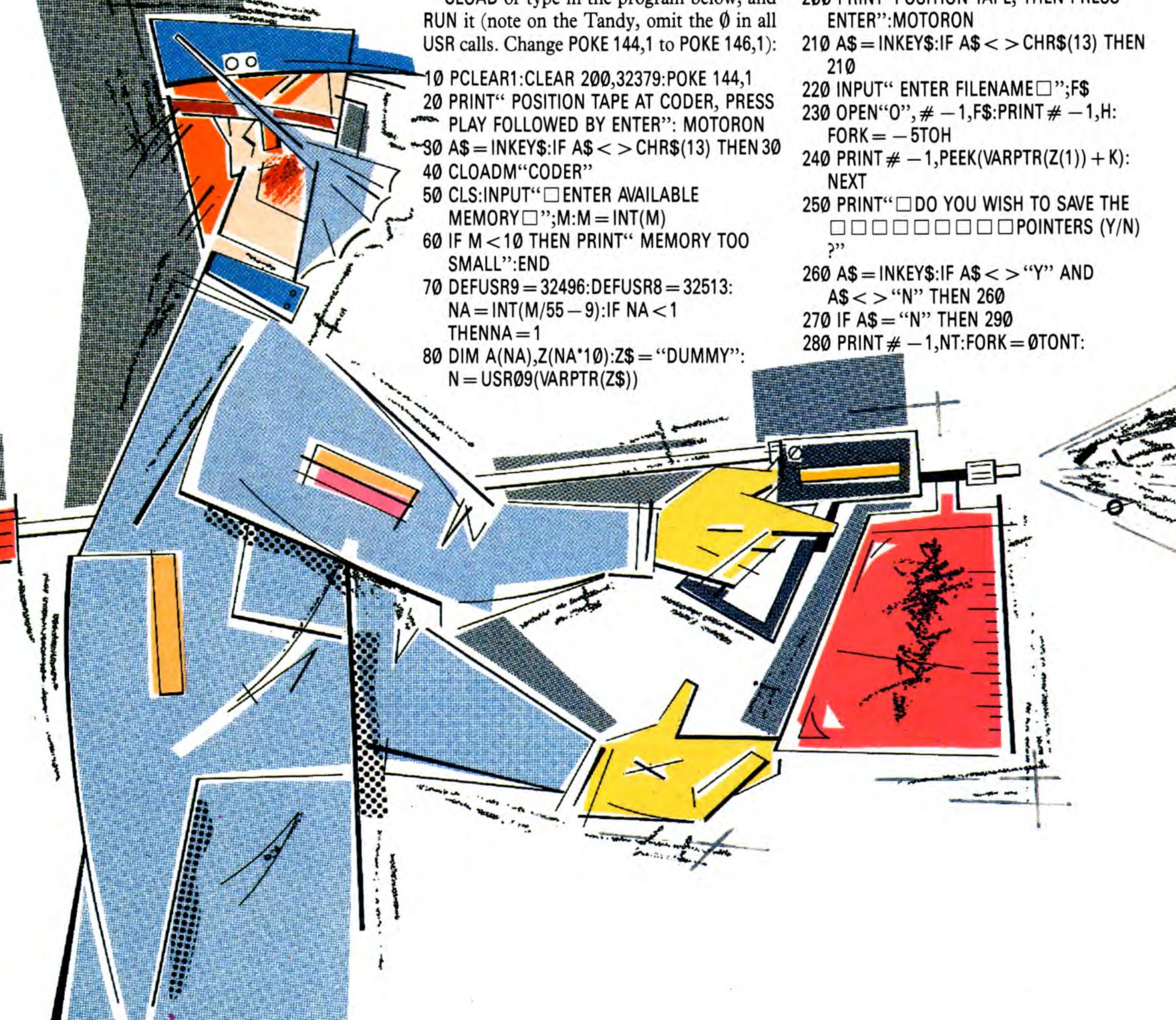
The number 5 is the variable N, which is used in the PROCedure to select the right pointers from a% which then pick the right message out of z%. SAVE the completed adventure on tape before you attempt to RUN it. When you do RUN the program to play the game or test it, have the text decoder and compressed text cassettes to hand ready to LOAD them in.

Having debugged your adventure containing the brief PRINT statements, you can now think about starting to use the text compressor.

First, make sure that you have RUN the adventure so that all the variables have been set up. Now find out how much memory remains—see page 268—and write the figure down.

CLOAD or type in the program below, and RUN it (note on the Tandy, omit the Ø in all USR calls. Change POKE 144,1 to POKE 146,1):

```
1Ø PCLEAR1:CLEAR 2ØØ,32379:POKE 144,1
2Ø PRINT" POSITION TAPE AT CODER, PRESS
    PLAY FOLLOWED BY ENTER": MOTORON
3Ø A$=INKEY$:IF A$<>CHR$(13) THEN 3Ø
4Ø CLOADM"CODER"
5Ø CLS:INPUT"□ENTER AVAILABLE
    MEMORY□";M:M=INT(M)
6Ø IF M<1Ø THEN PRINT" MEMORY TOO
    SMALL":END
7Ø DEFUSR9=32496:DEFUSR8=32513:
    NA=INT(M/55-9):IF NA<1
    THENNA=1
8Ø DIM A(NA),Z(NA*1Ø):Z$="DUMMY":
    N=USRØ9(VARPTR(Z$))
9Ø CLS:PRINT" INPUT TEXT NUMBER:";
    NT+1
1ØØ LINEINPUTZ$:IF Z$="" THEN 15Ø
11Ø A(NT)=USRØ8(VARPTR(Z(1)))
12Ø IF A(NT)>NA*5Ø THEN PRINT
    " AVAILABLE STORAGE EXCEEDED":END
13Ø NT=NT+1:IF NT>NA THEN PRINT
    " ALL POINTERS USED ":FORK=1
    TO1ØØØ:NEXT:GOTO15Ø
14Ø GOTO 9Ø
15Ø CLS:H=A(NT-1)
16Ø IF PEEK(H+VARPTR(Z(1)))<>Ø THEN
    H=H+1:GOTO16Ø
17Ø CLS:PRINT" SAVE DATA TO TAPE
    (Y/N) ?"
18Ø A$=INKEY$:IF A$<>"Y"
    ANDA$<>"N" THEN 18Ø
19Ø IF A$="N" THEN 3ØØ
2ØØ PRINT"POSITION TAPE, THEN PRESS
    ENTER":MOTORON
21Ø A$=INKEY$:IF A$<>CHR$(13) THEN
    21Ø
22Ø INPUT" ENTER FILENAME□";F$
23Ø OPEN"O",#-1,F$:PRINT#-1,H:
    FORK=-5TOH
24Ø PRINT#-1,PEEK(VARPTR(Z(1))+K):
    NEXT
25Ø PRINT"□DO YOU WISH TO SAVE THE
    □□□□□□□□□□POINTERS (Y/N)
    ?"
26Ø A$=INKEY$:IF A$<>"Y" AND
    A$<>"N" THEN 26Ø
27Ø IF A$="N" THEN 29Ø
28Ø PRINT#-1,NT:FORK=ØTONT:
```



```
   PRINT#—1,A(K):NEXT
290 CLOSE#—1
300 CLS:PRINT"NUMBER OF TEXT ARRAY
   ELEMENTS REQUIRED =";1+INT
   (H/5)
310 PRINT:PRINT"NUMBER OF POINTER
   ARRAY ELEMENTS REQUIRED =";NT
320 PRINT:PRINT"DO YOU A HARD COPY OF
   THE□□□□□□□POINTERS (Y/N) ?"
330 A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 330
340 IF A$="N" THEN 150
350 FORK=ØTONT—1:PRINT#—2,
   "POINTER□";K,A(K):NEXT
360 GOTO 150
```

The program allows you to enter your list of messages into the text compressor, but first it CLOADs in the coder.

RUN the program, and follow the instructions for CLOADing the coder. Next, enter the available memory, which you noted down after RUNning the adventure. There will be a prompt for each message. You should enter each of your messages, making sure that you don't use a carriage return in the middle, because the machine will interpret it as the message ending. If you want to make a message more than one line long, fill up the lines with spaces, which, incidentally, take up very little memory space.

If there's enough memory for all of your messages, the program asks you if you want to save the data on tape. If you're satisfied with your messages, you should select the Y option—make sure that you have a tape ready to receive the compressed text.

You have the option of not saving the pointers on tape, which will allow you to save even more memory. Unfortunately, you'll find that the next part of your adventure development is made a lot more fiddly. If you do decide not to save the pointers, you will need a list of their values. There's an option to make a hard copy of the pointers for those of you who have a printer.

If your messages occupy too much memory space, the computer will tell you AVAILABLE MEMORY EXCEEDED. Check how much of your text remains, which will give you some guide to how much you should shorten your messages—because of the way that the text is coded into varying length binary numbers, it's almost impossible to predict exactly by how much the messages have to be shortened. You may just be unlucky enough to find that the messages still do not fit, so it's safest to cut a little more than the number of letters would suggest.

When you need to RUN the program, type RUN 5Ø because the coder is already in memory, and it's a waste of time to CLOAD it again.

With the compressed text stored safely on tape, you can CLOAD back your adventure. Now add these lines to the beginning of the program—you may have to RENUMber your adventure to accommodate the extra lines.

```
10 CLEAR256,32379:DEFUSR7=32496:
   DEFUSR6=32507:POKE 144,1
20 CLOADM"DECODER"
30 DIMZ(106),A(21)
40 N=USR07(VARPTR(Z$))
50 OPEN"I",#—1,"FILENAME"
60 INPUT#—1,H:FORK=—5 TO H:
   INPUT#—1,N:POKE VARPTR(Z(Ø))
   +K,N:NEXT
70 INPUT#—1,NT:FORK=ØTONT:
   INPUT#—1,A(K):NEXT:CLOSE#—1
80 Z$="LENGTH OF LONGEST MESSAGE IN
   DUMMY CHARACTERS"
```



You should have the tape containing the decoder and the one containing the compressed text to hand when you RUN the adventure, for it needs to CLOAD them from tape to function correctly. You will probably need to change the FILENAME in Line 5Ø to the name of your compressed text.

Now you need to change all the PRINT statements for which you now have some compressed text. For example, if Line 1Ø5Ø is the tenth message and reads like this:

```
1050 PRINT" DARK ROOM "
```

it should be changed to:

```
1050 N=A(10):GOSUB9000:PRINTZ$
```

where the number in the brackets is the number of the message. You'll now need a subroutine like this so that the right part of the coded text will be selected and fed into Z$ to be PRINTed on the screen:

```
9000 DV=USR06(VARPTR(Z(1))+N):
   RETURN
```

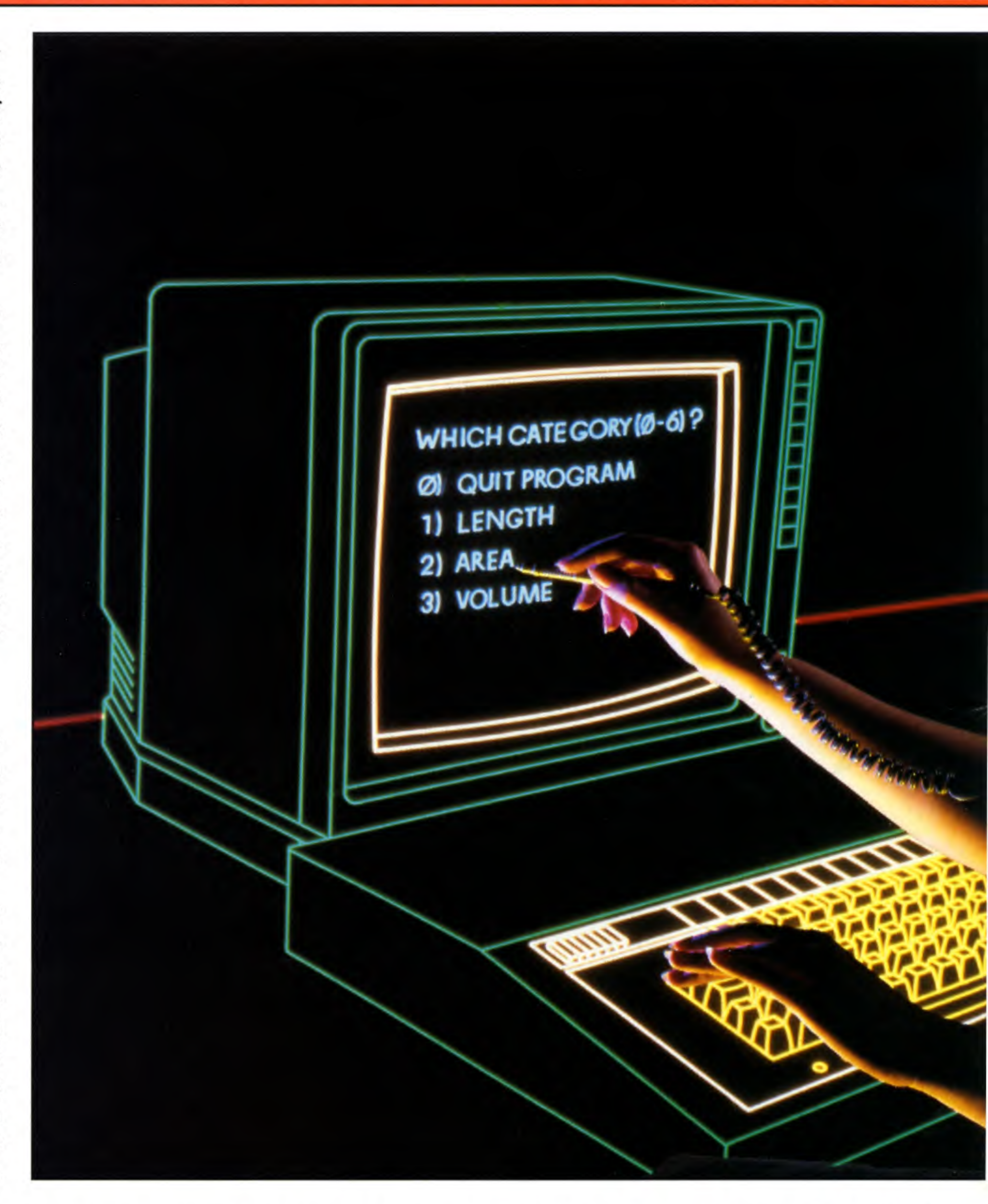
CSAVE the altered adventure program on tape ready for use.

TRIPPING THE LIGHT FANTASTIC

Pointing is a natural way to indicate what you want. Light pens give you an electronic pointer that you can use for anything from screen graphics to menu selection

Communicating with your computer usually means putting in the information through the keyboard, although there are a number of alternatives. Probably the commonest of these is a joystick—which you can use not just for games, but also for more serious applications. But another method that's not much more expensive and offers some distinct advantages is with a light pen.

Light pens are an exciting alternative to conventional control. As the name suggests, they're very similar to a conventional pen and you use them with a natural movement to 'draw' or otherwise indicate points on the screen. Because they can detect which position on the screen you are pointing to, they can, for instance, be used to make choices from a program that is directed by a menu. With conventional control, this would typically require you to take your eyes away from the screen and press one or more keys. With a light pen, you simply point at the option.



APPLICATIONS

Light pens are particularly suited to applications where you need to be able to locate a random position quickly on the screen. This may be to enter the coordinates of that position, to place a mark, draw a line, select from a menu or a host of other applications.

Light pens can be fast, easy to use, very accurate and are particularly suited to applications such as graphics. Here, using the cursor control (or even worse, entering coordinates via the keyboard), can be slow and laborious. With a light pen you can either draw 'freehand'—where you trace the outline you want drawn with the light pen—or you can point at the screen with the pen to choose various options from a graphics program, such as specifying the corners of a box.

Also, more and more games using light pens are appearing on the market. They tend to be of the type where objects are moved from one position to another.

Another development is a trend in the design of more serious applications programs, such as wordprocessors or accounts programs to adopt a 'you get what you see' approach. This means that instead of entering instructions by using control characters or function keys, they are entered by aligning the cursor with the desired item on the menu, in effect by pointing at the item.

Up until recently, applications for light pens have largely been left up to the users to work out for themselves. But now, with interest building up, more software writers are producing at least some programs with the light pen specifically in mind.

A number of programs available allow the pen to be used to create lines, circles, squares or other geometric shapes and fill them with various colours. Some graphics programs also allow the use of a light pen for freehand drawing. In conjunction with 'fill' commands these programs allow the user to create some inventive and intricate artwork.

One program, in particular, lets you use a light pen to define graphics characters, or symbols. You could then use a second



program to construct complicated pictures, in the same way as the articles on pages 484 to 491 and 528 to 533 do; the advantage is that you can use a light pen to select the various different options and characters, instead of having both to type in your choices with the keyboard and calculate screen coordinates.

Although light pens are not used all that much in games, which tend to use joystick control, some games, where objects on screen need to be moved between various points—for example chess, or draughts—do lend themselves well to light pen control.

Other games which you can buy for light pens include simple card games, word games, and maze games. In a maze game, for instance, you can use the light pen to point to objects which you want to pick up to help you to escape.

There is also a new development in the games field of light pens—the light rifle. This is already available for the Spectrum and Commodore computers, and is specifically designed for games. A variety of games software is already compatible with this rifle.

HOW LIGHT PENS WORK

You may have seen light pens on television or in the shops and apart from the lead, they look very like a large writing pen.

Essentially, though, a light pen 'reads' a position on a television or monitor screen. When it is pointed at the screen the electronics in the pen and the computer work together to calculate the coordinates of that point. This is possible because of the way in which the display on cathode ray tubes (the type used in televisions and monitors) is created. A tiny light dot scans the screen from left to right and from top to bottom. In the case of a British standard television set, it zig-zags down the 625 lines of the display 50 times in every second. This scanning is so fast that the eye cannot detect the movement and we perceive a stable image.

Light pens contain a light sensor in or near the tip. This sensor, which is usually either a light-sensitive transistor or diode, detects the scanning dot as it flies past. An electrical signal is produced, amplified and relayed to the computer. As it is the computer which controls the display on the screen, and therefore, 'knows' where the scanning dot will be on the screen from moment to moment it can then calculate the X and Y coordinates of the exact position of the tip of the light pen on the screen by comparing the moment of the impulse from the pen and the position of the dot.

Models of light pen differ in several ways: in the type of light sensor used; where the light sensor is placed (some, for example, channel the light to the sensor via a piece of optical fibre); where they house the associated electronics; whether they allow you to use different operating modes; whether, and if so how, they show you when they have picked up a signal; and last but not least, the type of computer with which they are compatible. These factors in turn affect the light pen's operating characteristics. Apart from price, the characteristics which distinguish a good light pen from a bad one are its sensitivity and resolution.

SENSITIVITY

A light pen's sensitivity refers to the range of screen light intensities which the pen can register. The better ones are able to pick up most, if not all of the colours displayed on a properly adjusted colour television screen. Lesser quality ones find it difficult to detect anything but the brightest colours. For many uses this is sufficient but it does impose a severe limitation on the flexibility of use of a

TROUBLESHOOTER

- If you find that your light pen behaves rather erratically, or is inaccurate, it may be picking up light from the room. You might need to shut the curtains, or turn off any very bright lamps.
- You may need to adjust the brightness and contrast controls on your television or monitor until the pen is able to work properly.
- When you use a light pen to make a choice from a menu, you should be careful not to wave the pen around—point it straight at the option you require; otherwise the pen could well detect a wrong dot.
- Some of the new light pens also detect which colour is on the screen; if you use one of these, make sure your software tells the pen which colours to detect. You may also need to adjust your TV or monitor's colour until it corresponds exactly with the light pen.

light pen, particularly in graphics applications where the drawing background may well be dark.

One problem with light pens is invalid detections. These occur when sensitivity is increased by a large amplification of the signal from the light sensor. Increasing sensitivity increases the likelihood of the pen being falsely triggered by light sources other than those of the screen—even daylight.

The more expensive light pens use a number of methods to ensure that only valid detections are made. The combination of light sensor and light filters can be chosen, for instance, closely to match the colour characteristics of the dot on the screen. Also, electronic filters can be used to ensure that the light source is actually from the screen. Light from the screen flickers 50 times a second. A simple electronic circuit (called a 'high-pass filter') is used to select only light that flickers at this frequency.

HIGH RESOLUTION PENS

Once the hurdles of sensitivity and validity are cleared, resolution can still be a problem. Resolution refers to the area of the point on the screen which is detected. This can vary from a single screen point or pixel, with a good light pen, to a bunch of characters with the worst.

A light pen's resolution depends on the type of sensor it uses, its speed of reaction and the method of 'collimation', or collecting light. Some light pens use an opaque tube to channel the light to the sensor, others use collimation optical lenses, or short lengths of fibre optics. High resolution is important if the pen is to be used for drawing rather than just detecting. A high resolution pen with good collimation can pinpoint a smaller dot, and so is more accurate.

Models of light pen differ in the amount of documentation and software which the manufacturer provides with them. For a light pen to work at all it requires support from software routines. The programming is fairly simple, whatever pen and computer you own. Future programs in *INPUT* will include routines using light pens.

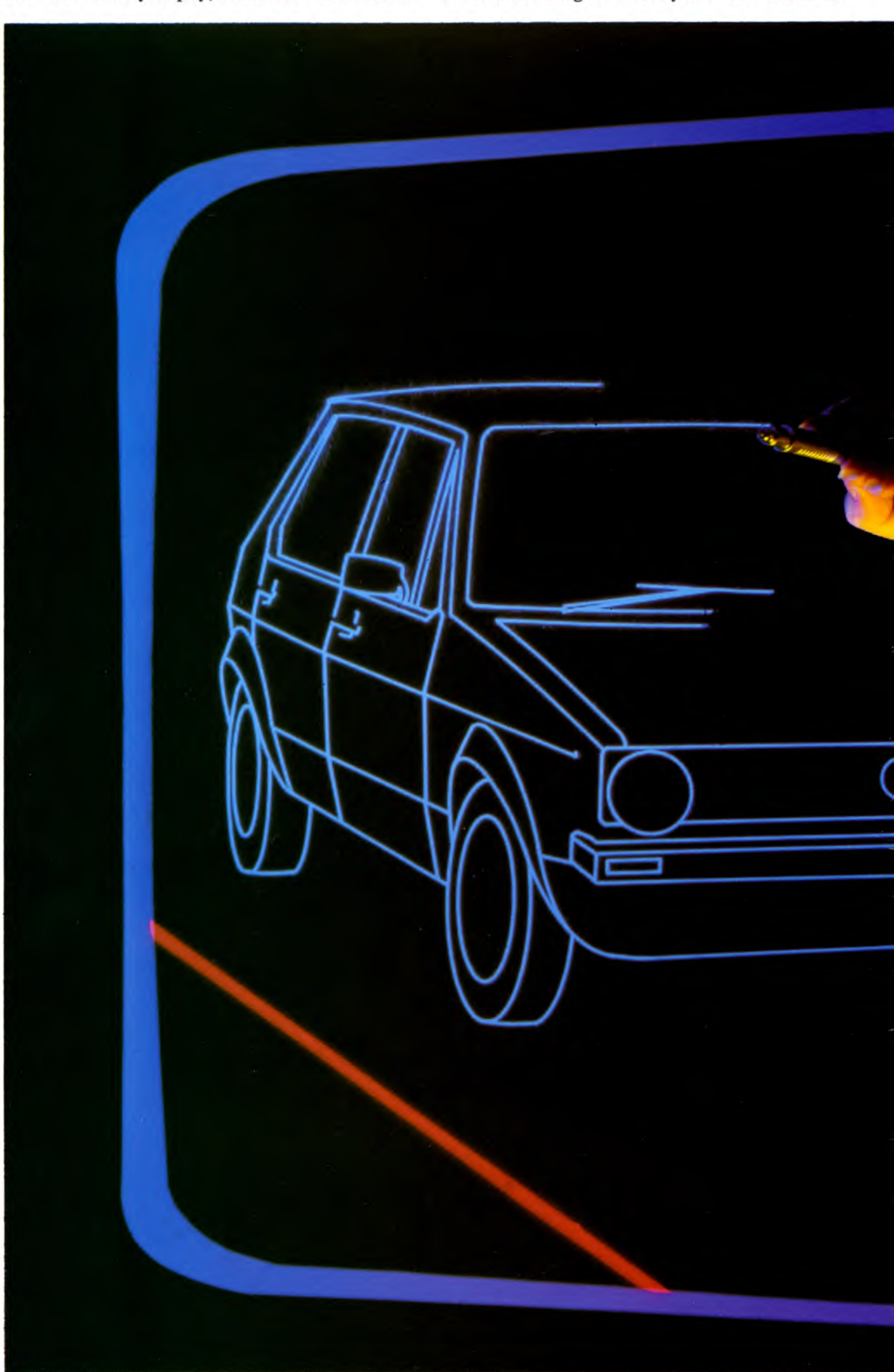
Because the pens need software to work, how useful and accurate they are depends largely upon how good the software is. Some manufacturers provide a cassette with example programs on, while others just provide listings to show you how to use the pen.

CHOOSING A LIGHT PEN

There are several factors to consider when selecting a light pen. For instance, you should take accuracy into account. While some light pens may have a high resolution in theory, they do not necessarily keep to it accurately. Some light pens are so inaccurate that, in a drawing program, lines drawn with them appear as a series of seemingly random pixels all over the screen.

With this, and all features of light pens, it seems to be a case of you get what you pay for—the more you pay, the more accurate the pen should be. Prices of light pens vary from the price of a typical memory expansion to the price of a cheaper home computer. However, as with most products associated with computing, prices are falling.

Some light pens, usually the more expensive ones, also provide two useful features not mentioned so far. The first of these features is some sort of signal to let you know when the



pen has detected a valid light dot. The second is some control that lets you determine when the signal is conveyed to the computer.

Light pens usually use a 'light-emitting-diode' to tell you when a valid dot has been detected. This feature is not essential if you just want the pen for drawing, but does help if you are using the pen to read positions from the screen, or make choices from a menu.

Pens often have a switch to let you send the information from the pen to the computer only when you want; this facility is useful for two main reasons: first, it reduces the chance of accidental, or false, readings (which would choose the wrong option on a menu, or plot the wrong dot). Second, it gives you more control over the readings taken by the pen. There are various types of switches used for this; some of the better pens, for example, have hoods or miniature buttons on the front, so that the switch is only turned on when the pen actually touches the TV or monitor screen.

When you are buying a light pen, you should try it out before you buy it, just to make sure it does what you want.



COMPATIBILITY

Another basic you must check before buying is whether or not it actually works with your computer. Like most other computer peripherals, light pens only work with the computer that they are made for.

You can sometimes get around this problem by connecting the pen to an interface; the Spectrum and ZX81, especially, almost certainly need an interface. If you buy a light pen specifically for your Sinclair computer, you may find some of these come with the interface built-in.

Don't forget that light pens need software to work. If you buy a pen that is not designed specifically for your computer, you might have to write or buy some software yourself to make it work.

PROBLEMS WITH LIGHT PENS

Light pens are prone to problems if they are not kept clean. Because they deal with light at fairly low intensity, any dirt on the front of the pen, or on the screen, can interfere with the light signal that the pen picks up. However, it's easy to clean the pen and/or the screen with one of many commercially available, alcohol-based cleaning kits.

Another problem peculiar to the light pen is that it obscures at least part of the screen when you use it. This is not too much of a handicap unless the program has a lot of detail on the screen.

Before buying, you must be sure that you can't do what you want more easily with a joystick. Don't forget that using a light pen for drawing can be very uncomfortable since the screen is vertical, as opposed to the normal action of drawing on a horizontal pad of paper.

Given the growing trend in both games and more serious applications to make the user interact more closely with the display, and the speed, ease of use, and the low cost of the light pen, it is bound to find increasing popularity. With that the price will drop and more software will become available with the light pen in mind. For the user who is looking for a cheap add-on that will allow a little more than just playing games, and enhance playing games to boot, a light pen is an ideal acquisition for the micro owner.

CHANCE AND PROBABILITY

Whether your interests lie in games strategy or in predicting life or death situations, don't rely on luck. Instead find out how to make well-informed guesses—and win

The power of computers lies in their ability to obey instructions repeatedly, accurately and at great speed. Compared with the almost instant thought processes of human beings, however, the performance of computers is much outclassed. The human brain is particularly good at making judgements and comparisons of parameters such as distance, speed and intensity of light.

But even this most sophisticated of organs fails pitifully when we try to judge the outcome of events. Yet it is essential to be able to say, for example, that, 'for insurance purposes, people are expected to live to 65 or 70 years', or that 'there is unlikely to be a major earthquake in the UK within the next century'. Such statements are common in everyday speech—we weigh up the *odds*—and they are also important for social, commercial and scientific purposes. When we use words such as *odds*, *prospects*, *doubt*, *expectations* and *likely*, we are making mental calculations of *probability*.

PROBABILITY EXPLAINED

Probability is a scientific measure of chance, and is used to judge the likely outcome of an event. It relies on there being a measurable number of outcomes, as in football matches, tossing coins, rolling dice, playing cards and playing fruit machines. Naturally, you should be able to measure or quantify the outcomes, so events such as horse racing and football matches are difficult subjects for probability. If you say 'I *expect* to win' then you are saying that you think there is a high *probability* that you'll win.

Most people rely on intuition as their main tool in calculating probability or chance. They guess. You can, however, give yourself a good edge in many matters involving chance by looking carefully at what the outcomes of any event might be, and which ones are more likely. You can learn to predict the probable results, and although there is no certainty about them, you are more likely to be right than someone who is making *uninformed* guesses.

PROBABILITY AND COMPUTING

So what has probability to do with computing? The answer is, quite a lot. Although the sort of probability mentioned above is very loose and dependent upon guesswork, it is possible to deduce mathematical rules with certain types of event which allow us to predict the likely result with a fair degree of accuracy.

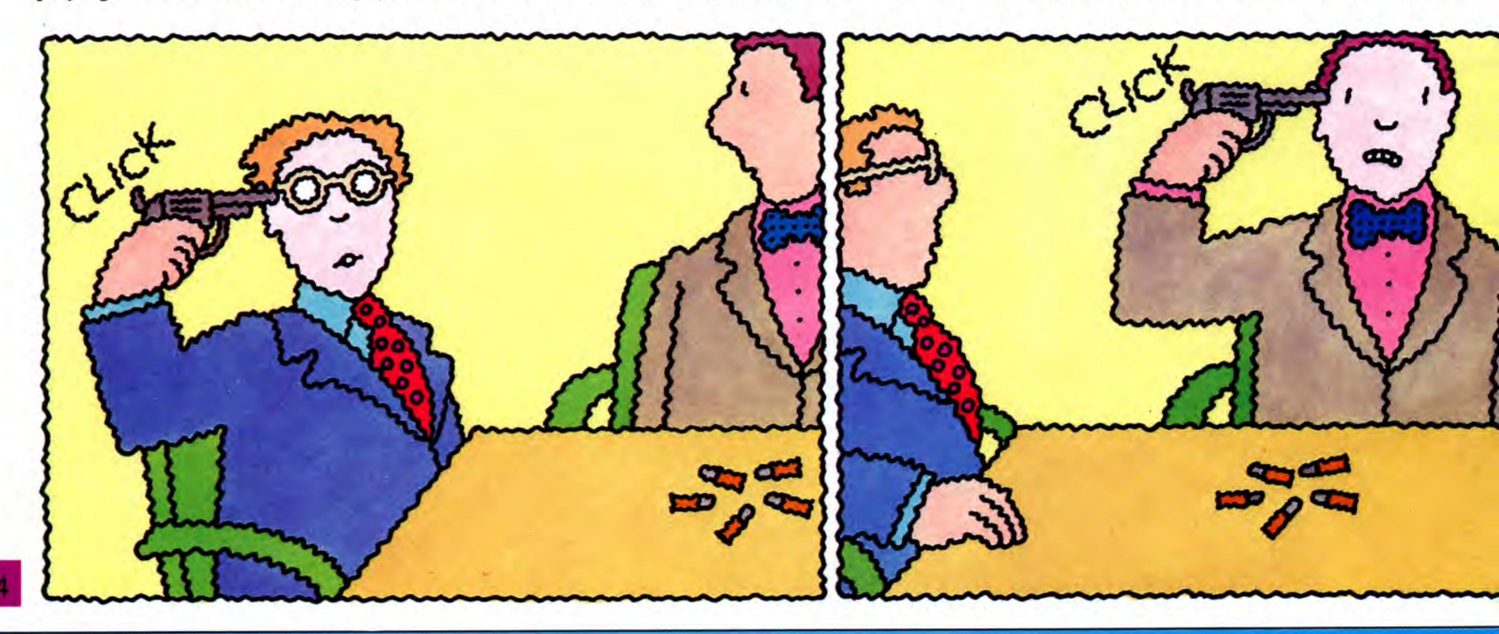
There are two ways in which computers can be useful here. In the first case, they can be used to simulate the event itself—it is a lot easier to get a computer to throw a die two thousand times to see what comes up than to do it yourself. Secondly, if you know the formulae for the expected outcome, you can also get the computer to calculate the result.

Depending upon your interests, this can be a purely theoretical exercise, or the basis of many useful applications of the computer. For an obvious application, look no further than games, many of which contain large elements of chance, where, for example, your score is dependent on the likelihood of particular results. But you could equally well write a program to tell you the chance of rain on any particular day (or the chance of a volcanic eruption in the UK!). For the moment, let's concentrate on the theory. Later on in *INPUT*, you'll see how you can set up some of these applications.

MEASURING PROBABILITY

The theory of measuring probability requires that you know the number of outcomes for any event, and that each of these occurs at some measurable frequency. The probability of any particular event occurring is the number of times it happens (its frequency) compared to the total of all the possible outcomes. In other words it is this frequency expressed as a fraction of all the possible outcomes.

Notice that if something is a certainty, its probability is 1. This is because if it happens



- WHAT IS PROBABILITY?
- HOW TO MEASURE IT
- PROBABILITY, FREQUENCY AND CHANCE
- COIN TOSSING PROGRAM
- PROBABILITY OF SEVERAL OUTCOMES
- PASCALS TRIANGLE
- FREQUENCY DISTRIBUTION
- PREDICTING THE OUTCOME

every time, its frequency equals the number of possible outcomes so when these are divided into one another, the result is 1. So 1 is the highest possible probability. And for all of several possible outcomes, their separate fractional probabilities will always total 1 when added together.

One of the simplest and oldest methods of reasoning probability is to spin a coin, and predict the result. Because the coin has two sides only, you know intuitively that in any number of tosses, you should get heads coming up half the time and tails the other half, ignoring the remote chance of the coin landing on its edge. To illustrate this method, key and RUN the first program (you need a Simons' BASIC cartridge for the Commodore 64, and a Super Expander cartridge for the Vic 20 for options 3 and 4):

```
5 BORDER 7: PAPER 7: INK 9: CLS
10 DIM n(4)
20 RESTORE 9000: FOR n=1 TO 4: READ
   n(n): NEXT n
30 INPUT "Which test (1-4)?",x: CLS
40 BORDER x: GOTO n(x)
50 REM Probability
60 REM Coin Toss
70 LET h=0: LET t=0
80 PRINT AT 2,6;"Press SPACE to toss"
90 PRINT AT 20,10;"HEADS: - □0";AT
   21,10;"TAILS: - □0"
100 IF INKEY$<>CHR$ 32 THEN GOTO
   100
110 IF x=2 THEN FOR n=1 TO 100
120 IF INT(RND*2)=1 THEN LET h=h+1:
   PRINT AT 10,15;"H";AT 20,18;h: GOTO
   130
125 LET t=t+1: PRINT AT 12,15;"T";AT
   21,18;t
130 IF x=1 THEN IF INKEY$<>CHR$ 32
   THEN GOTO 130
140 PRINT AT 10,15;"□";AT 12,15;"□"
150 IF x=1 THEN FOR m=1 TO 100: NEXT
   m: GOTO 120
155 NEXT n: STOP
9000 DATA 70,70,170,460
```

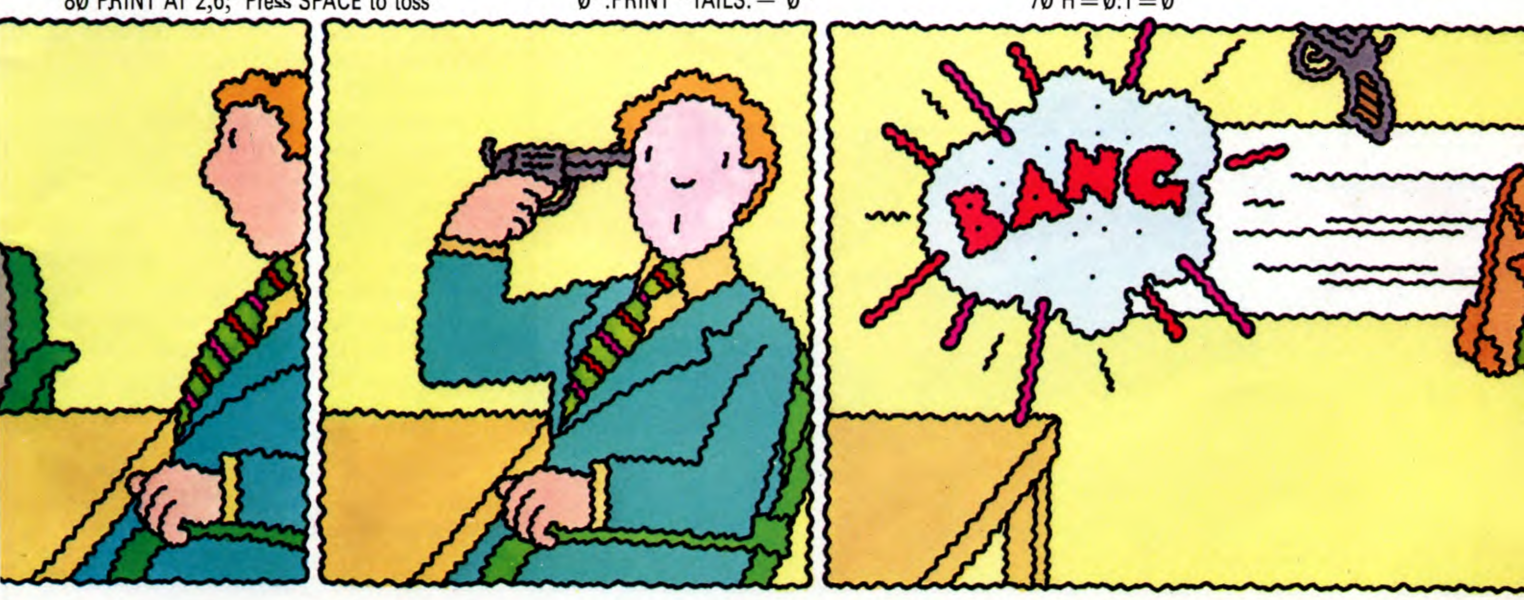
```
10 PRINT"[graphic][graphic]ENTER OPTION (1-4)"
20 POKE 53280,3: POKE 53281,1
30 INPUT X:PRINT "[graphic]":IF X<1
   OR X>4 THEN 10
40 ON X GOTO 70,70,180,460
50 REM ....PROBABILITY
60 REM ......COIN TOSS
70 H=0:T=0
80 PRINT "[graphic][graphic][graphic][graphic][graphic][graphic][graphic][graphic]
   PRESS THE SPACE BAR TO TOSS"
90 PRINT "[graphic][graphic][graphic][graphic][graphic]HEADS: -
   0":PRINT "TAILS: - 0"
100 GET A$:IF A$<>"□" THEN 100
110 IF X=2 THEN FOR N=1 TO 100
120 IF INT(RND(1)*2)+1=1 THEN
   H=H+1 :PRINT "[graphic][graphic][graphic][graphic]HEADS":
   GOTO 130
125 T=T+1 :PRINT "[graphic][graphic][graphic][graphic]
   TAILS"
130 PRINT "[graphic][graphic][graphic][graphic][graphic]";TAB(7);
   H:PRINT TAB(7);T
140 IF X=1 THEN GET A$:IF A$<>"□"
   THEN 130
150 IF X=1 THEN 120
160 NEXT N:END
```



The program is as for the Commodore 64, except these lines:

```
20 POKE 36879,27
80 PRINT "[graphic][graphic]HIT SPACE BAR
   TO TOSS."
```

```
10 MODE1
20 VDU23;8202;0;0;0;
30 PRINT:INPUT"WHICH TEST
   (1-4)□",X:CLS
40 ON X GOTO 70,70,180,460 ELSE 30
50 REM....................PROBABILITY
60 REM......................COIN TOSS
70 H=0:T=0
```



```
80 PRINT TAB(6,2)"PRESS THE SPACE BAR
   TO TOSS"
90 PRINT TAB(15,28)"HEADS: - □0"
   TAB(15,30)"TAILS: - □0"
100 G=GET:IF G<>32 THEN 100
110 IF X=2 THEN FOR N=1 TO 100
120 IF RND(2)-1 THEN H=H+1:PRINT
   TAB(19,15)"H" TAB(23,28);H: ELSE
   T=T+1:PRINT TAB(19,17)"T"
   TAB(23,30);T
130 IF X=1 THEN G=GET:IF G<>32
   THEN 130
140 PRINT TAB(19,15);"□" TAB(19,17);"□"
150 IF X=1 THEN FOR D=0 TO
   1000:NEXT:GOTO 120 ELSE NEXT
160 END
```

T

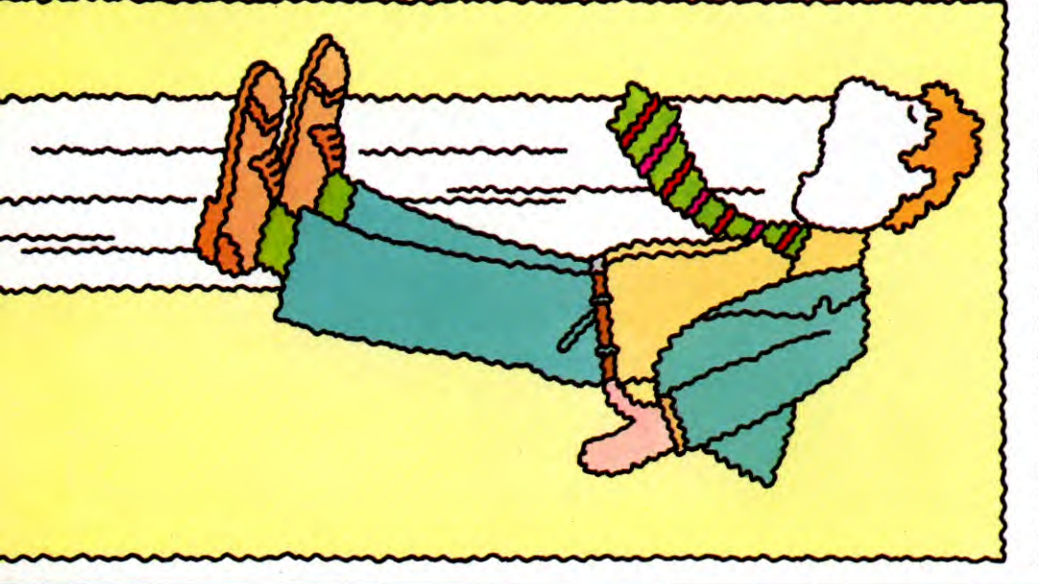
Users of the Tandy micro should change the 223 at Line 130 to 247.

```
10 PMODE3,1:CLS
20 INPUT"□WHICH TEST (1-4)□";X
30 CLS:IF X<1 OR X>4 THEN 20
40 ON X GOTO 70,70,180,460
50 REM...........PROBABILITY
60 REM.............COIN TOSS
70 H=0:T=0
80 PRINT@65,"PRESS THE SPACE BAR TO
   TOSS"
90 PRINT@288,"HEADS: - □0":PRINT
   "TAILS: - □0"
100 A$=INKEY$:IF A$<>"□" THEN 100
110 IF X=2 THEN FORN=1TO100
120 IFRND(2)-1 THENH=H+1:PRINT
   @204,"H":PRINT@295,H; ELSE
   T=T+1: PRINT@207,"T":
   PRINT@327,T;
130 IF X=1 AND PEEK(345)<>223 THEN
   130
140 PRINT@204,""
150 IF X=1 THEN FORD=0TO200:
   NEXT:GOTO120 ELSENEXT
160 A$=INKEY$:IF A$<>CHR$(13) THEN
   160 ELSEEND
```

This program will be developed through this article. When you RUN it you are prompted to enter a number to select a test. At this stage, you have entered only the first two tests, so enter 1, and you are ready to toss a coin—using the space bar or [SPACE]. The crux of the program is Line 120 which sets ones (Heads) or zeros (Tails) randomly. When a head is thrown, Line 120 PRINTs an H, but it PRINTs a T when a tail is thrown. The same line keeps a running count of the number of heads and tails as you toss repeatedly. Line 150 sets a delay (except on the Commodores) between tosses.

A few tosses will probably give very different values for H and T, but larger numbers of tosses will soon verify that they in fact occur with what becomes closer and closer to equal frequency—a half of the total number of tosses or 50/50. To demonstrate this fact on a large number of tosses, RUN the program again, but enter 2 to select the second test. This time when you press the space bar or [SPACE], Line 110 sets up a loop to toss the coin 100 times. Notice that the display shows H and T very close to 50. Change the 100 at Line 110 to 1000 and RUN, entering 2 again, and notice that both H and T are very close to 500.

Remember that although in a short test you could get heads every time you toss a coin, the probability of a head is always a half. You have to bear this in mind if you are looking at more than one event. Many people believe that if you toss a coin after getting ten tails in a row, then the chance of getting a head is greater than it would otherwise be. This is not the case. Past events cannot influence the fact that either a head or a tail is equally likely each time you throw. If you toss 11 coins at once, however, the probability of getting 11 tails is smaller than of getting ten tails and one head—although, in fact, it is even more likely that you will get close to an equal number of each.



MULTIPLE EVENTS

When there are several events, you need some additional information to be able to predict the probability of each outcome. One essential piece of information is the total of all possible outcomes. For example, if you toss a coin twice, there are three possible outcomes: two heads, a head and a tail, and two tails. You might think that each of these will occur a third of the time. In fact, the probabilities are two heads (1/4), two tails (1/4) and a head and a tail (1/2). To understand the third probability in the list, you need another essential piece of information—the number of occurrences of each outcome. A head and a tail occur twice, because there are *two* ways to get the result—a head then a tail, and a tail then a head—giving a total of four outcomes, three of which are different.

In practice, there are two mathematical tricks to spare you the effort of working out the number of each outcome. These are the binomial theorem and Pascal's Triangle. Binomial means consisting of two terms. If an event has only two possible outcomes, and you know the probability of each, then you can use the binomial theorem to give the probabilities.

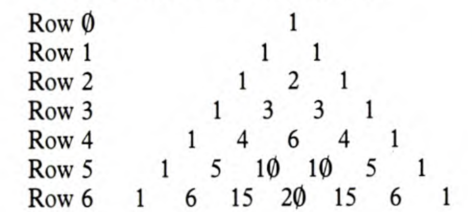
The binomial theorem tells us what to expect from repeated tests of an event with two outcomes. Call the probability of one outcome P and call the other outcome Q. (Remember that P and Q must add up to 1.) Call the number of events N.

In the example of tossing a coin, both P, the chance of a head, say, and Q, the chance of a tail, will be $\frac{1}{2}$, for one toss. According to the binomial theorem, the chance of any event occurring twice is the probability of it happening once, multiplied by itself. In general, the rule is that it is the probability of the event raised to the power N. So for two heads in two tosses, $P\uparrow N = \frac{1}{2} * \frac{1}{2} = \frac{1}{4}$. There is thus a one in four chance of getting two heads in a row. Similarly, for five in a row, $P\uparrow N$ gives $\frac{1}{2}\uparrow 5$, or 1/32.

As you will see later, you can use this method to calculate probability in any case where there are only two possible outcomes—the yes/no or head/tail instance. But what about the chance of something like getting

three heads and two tails out of five throws? For the answer to this, we need a more complex model.

A triangle of numbers devised by the French mathematician, Pascal, has many applications in mathematics, and it is equally useful here. It gives all the possible outcomes of any event with two results, and can be thought of as a number of rows of numbers. The first seven rows are as follows:



Row Ø	1
Row 1	1 1
Row 2	1 2 1
Row 3	1 3 3 1
Row 4	1 4 6 4 1
Row 5	1 5 1Ø 1Ø 5 1
Row 6	1 6 15 2Ø 15 6 1

To construct a triangle like this, write down the first two rows—Row Ø and Row 1—which are easily remembered. Row 2 then starts with a 1 to the left and ends with a 1 to the right of Row 1. The middle number (2) is obtained by adding the numbers in the row above (1 + 1). Similarly, Row 6 is obtained by adding 5 + 1, 5 + 1Ø, 1Ø + 1Ø, 1Ø + 5 and 5 + 1. By continuing this process, you can extend the triangle to a large number of rows, which would be difficult to figure otherwise.

Pascal's triangle gives all the information you need, if you are tossing several coins (or one coin several times). The number of coins gives the row to look at; the number of items in the row gives the number of different outcomes. For example, there are two outcomes for one coin (1 and 1 in Row 1) and seven for six coins (1, 6, 15, 2Ø, 15, 6 and 1 in Row 6). The sum of numbers in the row gives the total number of outcomes (2 for one coin, 4 for two coins and so on). Each number in the row gives the probability. For example, in Row 2, the first number (1) is the probability of two heads, the second number (2) is the probability of a head and a tail, and the third number (1) is for two tails. Of course, the frequency number has to be divided by the total number of outcomes (four, in this case) to give the probability. Notice that the result of adding up the numbers in each row is always a power of two (1, 2, 4, 8, 16). This is because for any one event, there are only two outcomes.

You can see how this method could be useful if you wish to calculate the probabilities of tossing, for example, 30 coins, but it would be very tedious in constructing the triangle to Row 30 and doing the calculations—besides the large space required to write the numbers. There is, however, a graphical method to deal with such cases, and here you can get the computer to help.

DISTRIBUTION CURVES

When there are many outcomes, and the probabilities seem unclear, you can often get good enough results to give you the answer you want by plotting a distribution curve. This is done by plotting the frequency of the results that you have on a graph—a frequency distribution. As with any graphical method, you can see at a glance much of the information it contains. If, for example, you were playing a game in which a coin is tossed 30 times, say (the same as tossing 30 coins at once), you could display the number of heads (or tails) from each set of 30 tosses. To see the result, key the next section of program, without erasing the first section:



```
170 REM Rnd Peaks
175 PLOT Ø,Ø: DRAW 18Ø,Ø
18Ø FOR x=4 TO 16Ø STEP 4
19Ø LET gm=Ø: GOSUB 61Ø
2ØØ FOR n=Ø TO h: PLOT x,n*6: NEXT n
22Ø NEXT x
23Ø STOP
61Ø REM Toss
62Ø LET h=Ø: LET t=Ø
63Ø PRINT AT 4,22;"HEADS:–";h;AT
    6,22;"TAILS:–";t
64Ø IF gm<>Ø THEN PRINT AT Ø,Ø;
    "GAMES:–";gm;AT 21,3;"HEADS FROM
    3Ø TOSSES:"
65Ø FOR s=1 TO 3Ø
66Ø IF RND>=.5 THEN LET h=h+1:
    PRINT AT 5,31;"H";AT 4,29;h;"□": GOTO
    67Ø
665 LET t=t+1: PRINT AT 5,31;"T";AT
    6,29;t;"□"
67Ø NEXT s
68Ø RETURN
```



```
17Ø REM ......RNDPEAKS
18Ø HIRES Ø,1:FOR X=Ø TO 3ØØ STEP 2Ø
19Ø GM=Ø:GOSUB 61Ø
```

```
200 FOR Y=0 TO H*6 STEP 6
210 TEXT X,200-Y,"*",1,1,8
220 NEXT Y,X
230 GOTO 230
610 REM TOSS
620 H=0:T=0
630 TEXT 220,10,"HEADS:",1,1,8
632 TEXT 220,20,"TAILS:"1,1,8
640 IFGM<>0THENTEXT 0,0,
   "GAMES:",1,1,8:TEXT 100,0,
   "HEADS FROM 30 TOSSES:",1,1,8
650 FOR TS=1 TO 30
660 IF INT(RND(1)*2)+1=1 THEN
   H=H+1 :GOTO 670
665 T=T+1
670 TEXT 263,10,STR$(H),1,1,8
672 TEXT 263,20,STR$(T),1,1,8
674 TEXT 263,10,STR$(H),0,1,8
676 TEXT 263,20,STR$(T),0,1,8
678 NEXT TS
680 RETURN
```

```
170 REM...... RNDPEAKS
180 GRAPHIC 2:FOR X=0 TO 1023 STEP 30
190 GM=0:GOSUB 610
200 FOR Y=0 TO H*34 STEP 34
210 POINT 1,X,995-Y
220 NEXT Y,X
230 GOTO 230
610 REM TOSS
620 H=0:T=0
630 CHAR 1,9,"HEADS:"
632 CHAR 2,9,"TAILS:"
640 IFGM<>0THEN:CHAR 0,9,
   "GAMES:":CHAR 3,1,"HEADS FROM 30:"
650 FOR TS=1 TO 30
660 IF INT(RND(1)*2)+1=1 THEN
   H=H+1 :GOTO 670
665 T=T+1
670 CHAR 1,15,"□□□"
672 CHAR 2,15,"□□□"
674 CHAR 1,15,STR$(H)
676 CHAR 2,15,STR$(T)
678 NEXT TS
680 RETURN
```



```
170 REM......................RNDPEAKS
180 FOR X=0 TO 1264 STEP 16
190 GM=0:PROCTOSS
200 FOR Y=0 TO H*30 STEP 10
210 PLOT69,X,Y
220 NEXTY:NEXTX
230 END
610 DEF PROCTOSS
620 H=0:T=0
630 PRINT TAB(28,4);"HEADS:□";H;
   "□□" TAB(28,6);"TAILS:□";T;"□□"
640 IF GM<>0 THEN PRINT
   TAB(2,9);"GAMES:□" TAB(13,30);
   "HEADS FROM 30 TOSSES:□"
650 FOR TS=1 TO 30
660 IF RND(2)-1 THEN H=H+1:
   PRINT TAB(39,5);"H" TAB(35,4);
   H ELSE T=T+1:PRINT TAB(39,5);
   "T" TAB(35,6);T
670 NEXT
680 ENDPROC
```





```
170 REM............RNDPEAKS
180 PCLS4:SCREEN1,0:FOR X=0 TO 255
   STEP 4
190 GM=0:GOSUB610
200 FOR Y=0 TO H*6 STEP 2
210 PSET(X,188-Y,2)
220 NEXT Y,X
230 GOTO160
610 H=0:T=0
650 FORTS=1 TO 30
660 IF RND(2)-1 THENH=H+1
   ELSET=T+1
670 NEXT
680 RETURN
```

Now RUN the program, entering 3 to select the third test. You should see a graph with a series of points ascending to various peaks on the screen. This is one of the many shapes that are possible with this type of analysis. The peaks are the number of heads from 30 tosses plotted along the Y-axis, and they are spaced out regularly along the X-axis. Notice that there are more higher than lower peaks. This is because the chance of you getting around 15, or about 12 to 17 heads are much higher than those for smaller or larger numbers of heads. You can see a similar pattern to this in the numbers in Pascal's triangle, with much larger values near the middle.

Line 180 sets up a loop to space out the peaks along the X-axis. The variable GM sets the number of 30-toss games to zero (Line 190) and a routine (Lines 610 to 680) is called to make each set of 30 tosses. This routine uses the elements of the second test, but it tosses the 'electronic' coin 30 times, instead of 100. Except on the Dragon and Tandy micros, as you run this test, you can notice the letters H and T (head and tail) coming up at the top right-hand corner of the screen. Once 30 tosses have been reached, the number of heads, which is accumulated in the routine, is scaled at Line 200 and PLOTted (Line 210) as Y-coordinates. The Dragon and Tandy cannot output text on a graphics screen, so this part of the display is omitted.

To get the most value from such an



analysis, you need to arrange the information to display one of the more recognizable statistical curves—a normal distribution. Key the next few lines to see a typical curve of this type:



```
45Ø REM Norm. Dist.
46Ø DIM g(3Ø)
47Ø PLOT 4,15Ø: DRAW Ø,−14Ø: DRAW
   245,Ø
48Ø GOSUB 56Ø
485 IF INKEY$< >CHR$32 THEN GOTO 485
56Ø REM Graph
57Ø PLOT 4,1Ø: FOR x=Ø TO 12ØØ STEP 2Ø
58Ø DRAW 4,6ØØ*FN n(ABS
   ((x−6ØØ)/14Ø))+1Ø−PEEK 23678
59Ø NEXT x: RETURN
6ØØ DEF FN n(x)=1/(PI*1.4142*
   2.718↑((x↑2)/2))
```

```
45Ø REM ........NORM DIST
46Ø HIRES Ø,1:DIM G(3Ø)
47Ø LINE Ø,Ø,Ø,2ØØ,1:LINE Ø,2ØØ,32Ø,2ØØ,1
48Ø GOSUB 56Ø
485 GET A$: IF A$< >"□" THEN 485
49Ø END
56Ø REM GR
565 DEF FN N(X)=1/(π*1.4142*
   2.718↑((X↑2)/2))
57Ø FOR X=1 TO 32Ø
58Ø PLOT X,199−(FNN((X−16Ø)/
   24)*53Ø),1
59Ø NEXT X: RETURN
```

```
45Ø REM.................NORM DIST
46Ø GRAPHIC 2: DIM G(3Ø)
47Ø DRAW 1,Ø,Ø, TO Ø,1Ø23 TO 1Ø23,
   1Ø23
48Ø GOSUB 56Ø
485 GET A$:IF A$< >"□" THEN 485
49Ø END
56Ø REM GR
565 DEF FN N(X)=1/(π*1.4142*
   2.718↑((X↑2)/2))
57Ø POINT 1,Ø,1Ø15:FOR X=1 TO 1Ø23
   STEP 8
58Ø DRAW 1 TO X.1Ø15−
   (FNN((X−511)/69)*3ØØØ)
59Ø NEXT X:RETURN
```

```
45Ø REM.....................NORM DIST
46Ø DIM G(3Ø)
47Ø MOVE 4Ø,7ØØ:DRAW 4Ø,1ØØ:
   DRAW 12ØØ,1ØØ
48Ø PROCGR
485 G=GET: IF G< >32 THEN 485
49Ø END
56Ø DEF PROCGR
57Ø GCOLØ,1:MOVE 4Ø,1ØØ:FOR
   X=4Ø TO 12ØØ STEP 8
58Ø DRAW X,3ØØØ*FNNORM
   ((X−6ØØ)/12Ø)+1Ø4
59Ø NEXT:GCOLØ,3:ENDPROC
6ØØ DEF FNNORM(X)=1/(PI*1.4142*
   2.718∧((X∧2)/2))
```

```
45Ø REM..............NORM DIST
46Ø DIM G(3Ø)
47Ø PCLS:SCREEN1,Ø:LINE(Ø,Ø)−
   (Ø,191),PSET:LINE−(255,191),PSET
48Ø GOSUB56Ø
485 A$=INKEY$:IF A$< >"□"
   THEN 485
56Ø DEFFND(X)=1/(4.4429*
   2.718↑((X*X)/2))
57Ø COLOR2,3:DRAW"BM2,19Ø":
   FORX=2 TO 255 STEP2
58Ø LINE−(X,191−64Ø*FND
   ((X−127)/24)),PSET
59Ø NEXT:RETURN
```

RUN the program and enter 4 to see an ideal normal-type distribution curve. Line 46Ø dimensions an array, which you need later to keep a count of heads thrown. Line 47Ø draws two X–Y coordinate axes, and Line 48Ø calls a routine to draw the curve. This routine uses a mathematical function (Line 58Ø) to draw the curve, which explains its perfect shape—all the points join up to give a smooth curve. The function is defined at Line 6ØØ (Line 56Ø on the Dragon and Tandy and Line 565 on the Commodore 64 and Vic 20 micros). Do not press any other keys yet, because the routine is incomplete, and will give an error.

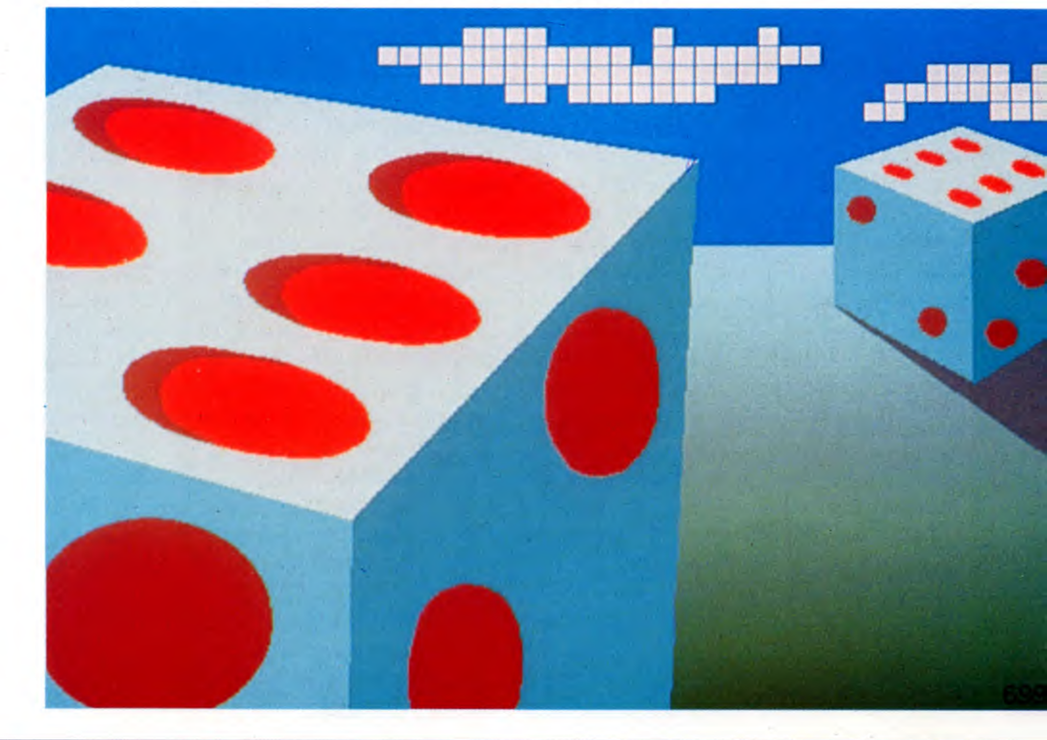
A smooth curve is very rarely obtained when the information is plotted from actual data. This is to be expected, because you are dealing with probabilities, and not certainties. The probability of an outcome—such as a shower of rain during the monsoon season in India—is high, but there have been periods when drought replaces the expected downpours. The next test illustrates this point well. What it does is to repeat the earlier coin tossing experiment many times and plot the results. Key the second part of the fourth test:



```
49Ø FOR g=1 TO 2ØØ: LET gm=g
5ØØ GOSUB 61Ø
51Ø LET g(h)=g(h)+1
52Ø PLOT 8+8*h,1Ø+4*g(h)
53Ø PRINT AT 21,25;h;"□"
54Ø NEXT g
55Ø STOP
```



```
49Ø FOR GM=1 TO 2ØØ
5ØØ GOSUB 61Ø
5Ø2 TEXT 43,Ø,STR$(G1),Ø,1,8
5Ø4 TEXT 263,Ø,STR$(G2),Ø,1,8
51Ø G(H)=G(H)+1
52Ø PLOT H*1Ø+1Ø,2ØØ−G(H)*4,1
53Ø TEXT 43,Ø,STR$(GM),1,1,8:G1=GM
532 TEXT 263,Ø,STR$(H),1,1,8:G2=H
54Ø NEXT GM
55Ø GOTO 55Ø
```



```
49Ø FOR GM=1 TO 2ØØ
5ØØ GOSUB 61Ø
5Ø2 CHAR Ø,15,"□□□"
5Ø4 CHAR 3,15,"□□□"
51Ø G(H)=G(H)+1
52Ø POINT 1,H*34,1Ø23-G(H)*2Ø
53Ø CHAR Ø,15,STR$(GM)
532 CHAR 3,15,STR$(H)
54Ø NEXT GM
55Ø GOTO 55Ø
```

```
49Ø FOR GM=1 TO 5ØØ
5ØØ PROCTOSS
51Ø G(H)=G(H)+1
52Ø PLOT69,4Ø*H,1Ø*G(H)+1ØØ
53Ø PRINT TAB(35,3Ø);H;"□"
   TAB(9,9);GM
54Ø NEXT GM
55Ø END
```

```
49Ø FOR GM=1TO5ØØ
5ØØ GOSUB61Ø
51Ø G(H)=G(H)+1
52Ø PSET(H*8+7,192-G(H)*2,3)
53Ø NEXT
54Ø GOTO 16Ø
```

Now RUN the fourth test again. When the ideal curve has been drawn, press the space bar or SPACE to start tossing. Then notice a series of points 'grow' to fill the space within the curve. When the test is complete, 500 (200 on the Spectrum and Commodores) points will have been PLOTted (set at Line 49Ø). On some micros, such as the Spectrum, the running time for this test is several minutes. This is why the Spectrum and Commodore perform the test fewer times by using 200 rather than 500 in Line 49Ø. Users of the Commodore 64 can speed up the execution of this test further by entering GOTO 64Ø at Line 625 and GOTO 678 at Line 667, but you must delete these changes to run the other tests.

This section of the program calls the routine (Line 5ØØ) that tosses a coin 30 times so, as in the third test, the tosses are flashed at the top, right-hand side of the screen (except on the Dragon and Tandy). Each set of 30 tosses is a game, and can result in any number of heads between 0 and 30. So in the array at Line 46Ø, H can vary between Ø and 3Ø. The Spectrum does not have Ø as the first variable in an array—as do other micros such as the Acorns—so some provision could be made for the rare event when no heads result from a game of 30 tosses. But this is likely to be such a rare occurrence because of the way that the random numbers are generated, that no provision is made here. In fact, extremely small or extremely large numbers of heads out of 30 tosses are most unlikely—their occurrence is *possible*, but the probability of this happening is very low.

Using the array, Line 51Ø keeps a count of the results of each game. For example, every time the result of a game is 11 heads, the array G(11) is increased by one. Similarly, every time the result is 15 heads, G(15) is increased by one. At the start, all array variables are Ø.

After each game, Line 52Ø scales the value of H (the number of heads for 30 tosses) to yield X and Y coordinates. The next time the same result occurs, a point is plotted at the same X-position, but one unit farther up the Y-axis. Line 53Ø keeps a count of H for each game, as well as a count of the number of games.

USING THE CURVE

Run the fourth test a few times to see how the profile of points within the curve varies, then do the same again, but with smaller end values for GM at Line 49Ø. Even without the ideal curve, you will soon be able to imagine an idealized curve through the peaks. In practice, however, the reverse of this imaginary process is of far greater value—if you know the profile of the curve, you can predict the results of future tests.

The value of H at the central peak is of special interest. It is the mean, or mathematical average, of the 31 possible H-values along the X-axis. In this case, it is 15. The mean identifies the peak of the curve. This is the most likely single value, but by itself it is not a particularly useful piece of information. Although you can say that 15 is most likely, 14 or 16 are only slightly less likely. There's a number of common values around the peak, and it is also useful to know how widely these are spread. So the mean is used to specify another important statistical parameter—the standard deviation—a measure of the spread. The formula for standard deviation is complicated, but not without good reason. Once you have calculated this parameter, you can assign a probability to any point on the curve.

The standard deviation is a measure of how much the values vary on both sides of the mean. For example, a section of the curve with a spread of 1.96 on either side of the mean will enclose 95 per cent of the results. If you extend the standard deviation to 2.58, the curve will include 99 per cent of the results. If you use commercial statistics software packages, they will have the facility to calculate standard deviation.

A CASE OF SIX OUTCOMES

There are many instances of events that have more than the two possible outcomes in the simple example of the coin throwing. In such cases, working out the likelihood of any one event is not as easy as picking the relevant row of Pascal's Triangle. For example, in the case of dice, when you roll one die there are six possible outcomes. Providing you are using balanced dice, the six results are equally likely. The results from two dice can be worked out by drawing up a table, but inevitably, the more results you have the more complicated the method of working them out you have to follow becomes.

Here is a table of all the results possible from rolling a pair of dice:

		first dice value					
		1	2	3	4	5	6
	1	2	3	4	5	6	7
second	2	3	4	5	6	7	8
dice	3	4	5	6	7	8	9
value	4	5	6	7	8	9	1Ø
	5	6	7	8	9	1Ø	11
	6	7	8	9	1Ø	11	12

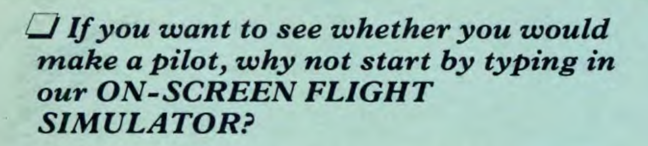
As you can see in the table, there are 36 possible ways the dice can fall—six rows times six columns—although there are only 11 different results. Several other facts are apparent from this table. There is only one chance in 36 of getting either the lowest or the highest score (2 or 12 occur only once in the table), whereas there are six chances in 36 (1/6) of getting a score of seven. There are also six chances in 36 of throwing a double. These are given by the diagonal numbers 2 (two ones) to 12 (two sixes).

With a combination of the Binomial theorem and this table, you can work out the probabilities of multiple events. A good example of a multiple dice throw event can be taken from the game of Monopoly. If you end up in jail, you have three goes to try and throw a double, otherwise you have to pay the fine. Intuitively, it appears that you have a 50/50 chance of getting off (3 goes, and 1/6 chance each time), but this is not the case. From the table, you can see that the chance of *not* throwing a double each time is 30/36 (5/6). Using the binomial theorem, you can see that the chance of not throwing a double three times in a row is 5/6 to the power of 3, or 125/216. This is about 58 per cent chance of failure, so if you can afford to, and you want to get out of jail, it pays to pay the fine and leave jail without further ado.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

COMING IN ISSUE 23...



❑ *If you want to see whether you would make a pilot, why not start by typing in our ON-SCREEN FLIGHT SIMULATOR?*

❑ *Fed-up with working out UDGs on paper? There's a handy SCREEN PLANNER PROGRAM to spare you all the effort*

❑ *Learn more about music generation, including how to TRANSCRIBE YOUR FAVOURITE TUNES onto the computer*

❑ *Find out how you can link up to other enthusiasts via the telephone system and a computer BULLETIN BOARD*

❑ *Discover more of the theory and practice of SORTING METHODS, with routines for even greater speeds*

❑ *PLUS ... for SPECTRUM users, a guide to the subtle SOUND EFFECTS possible in MACHINE CODE*

A MARSHALL CAVENDISH 23 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

ASK YOUR NEWSAGENT FOR INPUT